IBM ILOG Solver

Reference Manual

June 2009

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

About This Manual

This reference manual documents the IBM® ILOG® Concert Technology and IBM ILOG Solver libraries.

Group Summary	
optim.concert	The IBM® ILOG® Concert API.
optim.concert.extensions	The IBM® ILOG® Concert Extensions Library.
optim.concert.solver	The IBM® ILOG® Concert Solver API.
optim.concert.xml	The IBM ILOG Concert Serialization API.
optim.solver	The IBM® ILOG® Solver API.
optim.solver.iim	The IBM® ILOG® Iterative Improvement Methods Optimization Components (IIM) API.

What is Solver?

Solver is a C++ library for *constraint programming*. This library is not a new programming language: it lets you use data structures and control structures provided by C++. Thus, the Solver part of an application can be completely integrated with the rest of that application (for example, the graphic interface, connections to databases, etc.) because it can share the *same* objects.

What Is Concert Technology?

Concert Technology offers a C++ library of classes and functions that enable you to design models of problems for both math programming (including linear programming, mixed integer programming, quadratic programming, and network programming) and constraint programming solutions.

This library is not a new programming language: it lets you use data structures and control structures provided by C++. Thus, the Concert Technology part of an application can be completely integrated with the rest of that application (for example, the graphic interface, connections to databases, etc.) because it can share the same objects.

Furthermore, you can use the same objects to model your problem whether you choose a constraint programming or math programming approach. In fact, Concert Technology enables you to combine these technologies simultaneously.

What You Need to Know

This manual assumes that you are familiar with the operating system where you are using Solver. Since Solver is written for C++ developers, this manual assumes that you can write C++ code and that you have a working knowledge of your C++ development environment.

Notation

Throughout this manual, the following typographic conventions apply:

- Samples of code are written in this `typeface`.
- The names of constructors and member functions appear in this `typeface` in the section where they are documented.
- Important ideas are emphasized like *this*.

Naming Conventions

The names of types, classes, and functions defined in the Solver library begin with `Ilo` or `Ilc`. Names beginning with `Ilo` (for optimization) indicate components suitable for models; they are recommended for use with IBM ILOG Concert Technology. Names beginning with `Ilc` (for constraint programming) indicate components suitable for use inside a search; they are necessary when you extend the Solver library yourself by nesting search components within goals or constraints or by deriving your own classes from `Ilc` components.

The names of classes are written as concatenated, capitalized words. For example:

`IlcIntVar`

A lower case letter begins the first word in names of arguments, instances, and member functions. Other words in such a name begin with a capital letter. For example,

`aVar`

`IlcIntVar::getValue`

There are no public data members in Solver except in goals and demons. (See for an explanation of goals and demons.)

Accessors begin with the keyword `get` followed by the name of the data member. Accessors for Boolean members begin with `is` followed by the name of the data member. Like other member functions, the first word in such a name begins with a lower case letter, and any other words in the name begin with a capital letter.

Modifiers begin with the keyword `set` followed by the name of the data member.

```
class Task {
public:
    Task(char* name, IlcInt duration);
    ~Task();
    IlcInt getDuration() const;
    void setDuration(IlcInt duration);
    IlcBool isCritical() const;
    void setCritic(IlcBool critic);
};
```

Concepts

Arrays

For most basic classes (such as `IloNumVar` or `IloConstraint`) in Concert Technology, there is also a corresponding class of arrays where the elements of the array are instances of that basic class. For example, elements of an instance of `IloConstraintArray` are instances of the class `IloConstraint`.

Arrays in an Environment

Every array must belong to an environment (an instance of `IloEnv`). In other words, when you create a Concert Technology array, you pass an instance of `IloEnv` as a parameter to the constructor. All the elements of a given array must belong to the same environment.

Extensible Arrays

Concert Technology arrays are extensible. That is, you can add elements to the array dynamically. You add elements by means of the `add` member function of the array class.

You can also remove elements from an array by means of its `remove` member function.

References to an array change whenever an element is added to or removed from the array.

Arrays as Handles

Like other Concert Technology objects, arrays are implemented by means of two classes: a handle class corresponding to an implementation class. An object of the handle class contains a data member (the handle pointer) that points to an object (its implementation object) of the corresponding implementation class. As a Concert Technology user, you will be working primarily with handles.

Copying Arrays

Many handles may point to the same implementation object. This principle holds true for arrays as well as other handle classes in Concert Technology. When you want to create more than one handle for the same implementation object, you should use either the copy constructor or the assignment operator of the array class. For example,

```
IloNumArray array(env);     // creates a handle pointing to new impl
IloNumArray array1(array);  // creates a handle pointing to same impl
IloNumArray array2;         // creates an empty handle
array2 = array;        // sets impl of handle array2 to impl of array
```

Programming Hint: Using Arrays Efficiently

If your application only reads an array (that is, if your function does not modify an element of the array), then we recommend that you pass the array to your function as a `const` parameter. This practice forces Concert Technology to access the `const` conversion of the index operator (that is, `operator[]`), which is faster.

Assert and NDEBUG

Most member functions of classes in Concert Technology are inline functions that contain an `assert` statement. This statement asserts that the invoking object and the member function parameters are consistent; in some member functions, the `assert` statement checks that the handle pointer is non-null. These statements can be suppressed by the macro `NDEBUG`. This option usually reduces execution time. The price you pay for this choice is that attempts to access through null pointers are not trapped and usually result in memory faults.

Compilation with `assert` statements will not prevent core dumps by incorrect code. Instead, compilation with `assert` statements moves the execution of the incorrect code (the core dump, for example) to a place where you can see what is causing the problem in a source code debugger. Correctly written code will never cause one of these Concert Technology `assert` statements to fail.

Column-Wise Modeling

Concert Technology supports column-wise modeling, a technique widely used in the math programming and operations research communities to build a model column by column. In Concert Technology, creating a new column is comparable to creating a new variable and adding it to a set of constraints. You use an instance of `IloNumColumn` to do so. An instance of `IloNumColumn` allows you to specify to which constraints or other extractable objects Concert Technology should add the new variable along with its data. For example, in a linear programming problem (an LP), if the new variable will appear in some linear constraints as ranges (instances of `IloRange`), you need to specify the list of such constraints along with the non zero coefficients (a value of `IloNum`) for each of them.

You then create a new column in your model when you create a new variable with an instance of `IloNumColumn` as its parameter. When you create the new variable, Concert Technology will add it along with appropriate parameters to all the extractable objects you have specified in the instance of `IloNumColumn`.

Instead of building an instance of `IloNumColumn`, as an alternative, you can use a column expression directly in the constructor of the variable. You can also use instances of `IloNumColumn` within column expressions.

The following undocumented classes provide the underlying mechanism for column-wise modeling:

- `IloAddValueToObj`
- `IloAddValueToRange`

The following operators are useful in column-wise modeling:

- in the class `IloRange`,

```
IloAddValueToRange operator() (IloNum value);
```

- in the class `IloObjective`,

```
IloAddValueToObj operator () (IloNum value);
```

That is, the `operator ()` in extractable classes, such as `IloRange` or `IloObjective`, creates descriptors of how Concert Technology should add the new, yet-to-be-created variable to the invoking extractable object.

You can use the `operator +` to link together the objects returned by `operator ()` to form a column. You can then use an instance of `IloNumColumn` to build up column expressions within a programming loop and thus save them for later use or to pass them to functions.

Here is how to use an instance of `IloNumColumn` with operators from `IloRange` and `IloObjective` to create a column with a coefficient of 2 in the objective, with 10 in `range1`, and with 3 in `range2`. The example then uses that column when it creates the new variable `newvar1`, and it uses column expressions when it creates `newvar2` and `newvar3`.

```
IloNumColumn col = obj(2) + range1(10) + range2(3);
IloNumVar newvar1(col);
IloNumVar newvar2(col + range3(17));
IloNumVar newvar3(range1(1) + range3(3));
```

In other words, given an instance `obj` of `IloObjective` and the instances `range1`, `range2`, and `range3` of `IloRange`, those lines create the new variables `newvar1`, `newvar2`, and `newvar3` and add them as linear terms to `obj`, `range1`, and `range3` in the following way:

```
obj: + 2*newvar1 + 2*newvar2
range1: +10*newvar1 + 10*newvar2 + 1*newvar3
range2: + 3*newvar1 + 3*newvar2
range3: + 17*newvar2 +3*newvar3
```

For more information, refer to the documentation of `IloNumColumn`,`IloObjective`, and `IloRange`.

Lazy Copy

Concert Technology makes a lazy copy when you use any of the following objects inside a predefined Concert Technology object:

- an expression (an instance of `IloExpr` or one of its subclasses),
- a column (an instance of `IloNumColumn`),
- or a set (such as an instance of `IloIntSet`).

That is, a physical copy of those objects is created only when needed.

In Concert Technology, expressions, columns, and sets are implemented by handle classes and corresponding implementation classes. One or more handles may point to the same implementation object. For example, many instances of the handle class `IloNumColumn` may point to the same implementation object.

A handle may be empty; that is, it may point to 0 (zero). You can test whether a handle is empty by means of the member function `handle.getImpl`. If that member function returns 0, the handle points to a null implementation.

When you modify an expression, a column, or a set that has been used in a Concert Technology object, Concert Technology considers whether the handle you are modifying is the sole reference to the corresponding implementation object. If so, Concert Technology simply makes the modification.

In contrast, if the handle you are modifying points to an implementation object that is used by other objects predefined in Concert Technology, Concert Technology first copies the implementation object for the handle you are modifying and then makes the modification. The other handles pointing to the original implementation object remain unchanged and your modification has no impact on them.

Examples:

Here is an example illustrating lazy copy of variables:

```
IloEnv env;
IloIntVar a1(env, 0, 10);
IloIntVar a2(env, 0, 10);
IloIntVar a3(env, 0, 10);
IloExpr e = a1+ a2;;
IloConstraint ct = e <= 10;
e += a3;
return 0;
```

Because of the lazy copy, even though `a3` was added to `A`, `ct` uses only `a1` and `a2`.

Normalization: Reducing Linear Terms

Normalizing is sometimes known as *reducing the terms* of a linear expression.

Linear expressions consist of terms made up of constants and variables related by arithmetic operations; for example, x + 3y is a linear expression of two terms consisting of two variables. In some expressions, a given variable may appear in more than one term, for example, x + 3y +2x. Concert Technology has more than one way of dealing with linear expressions in this respect, and you control which way Concert Technology treats expressions from your application.

In one mode, Concert Technology analyzes linear expressions that your application passes it and attempts to reduce them so that a given variable appears in only one term in the linear expression. This is the default mode. You set this mode with the member function `IloEnv::setNormalizer(IloTrue).`

In the other mode, Concert Technology assumes that no variable appears in more than one term in any of the linear expressions that your application passes to Concert Technology. We call this mode assume normalized linear expressions. You set this mode with the member function `IloEnv::setNormalizer(IloFalse).`

In classes such as `IloExpr` or `IloRange`, there are member functions that check the setting of the member function `IloEnv::setNormalizer` in the environment and behave accordingly. The documentation of those member functions indicates how they behave with respect to normalization.

When you set `IloEnv::setNormalizer(IloFalse)`, those member functions assume that no variable appears in more than one term in a linear expression. This mode may save time during computation, but it entails the risk that a linear expression may contain one or more variables, each of which appears in one or more terms. Such a case may cause certain assertions in member functions of a class to fail if you do not compile with the flag `-DNDEBUG`.

By default, those member functions attempt to reduce expressions. This mode may require more time during preliminary computation, but it avoids of the possibility of a failed assertion in case of duplicates.

For more information, refer to the documentation of `IloEnv`,`IloExpr`, and `IloRange`.

Notification

You may modify the elements of a model in Concert Technology. For example, you may add or remove constraints, change the objective, add or remove columns, add or remove rows, and so forth.

In order to maintain consistency between a model and the algorithms that may use it, Concert Technology notifies algorithms about changes in the objects that the algorithms have extracted. In this manual, member functions that are part of this notification system are indicated like this:

Note
This member function notifies Concert Technology algorithms about this change of this invoking object.

Deletion of Extractables

As a modeling layer, Concert allows the creation and destruction of extractables. This is accessible through the method `IloExtractable::end()` and `IloExtractableArray::endElements()` method. The goal of these methods is to reclaim memory associated with the deleted objects while maintaining the safest possible Concert environment. In this context, a safe Concert environment is defined by the property that no object points to a deleted object; this is referred to as a dangling pointer in C++.

There exist two paradigms to ensure the safeness of the delete operation. The first, linear mode, comes from math programming and is possible only on extractables and objects used in linear programming. The second, safe generic mode, is more strict and is valid on all Concert extractables.

You can access both paradigms by calling `IloEnv::setDeleter(IloDeleterMode mode)`, where mode may be `IloLinearDeleterMode` or `IloSafeDeleterMode`.

Linear Mode

To use linear mode, you must either

- call `IloEnv::setDeleter(IloLinearDeleterMode)`, or

- refrain from calling `IloEnv::setDeleter()`, as it is the default mode.

In linear mode, the following behavior is implemented:

- If a range constraint is deleted, it is removed from the models that contain it.
- If a variable is deleted, its coefficient is set to `0` in the ranges, expressions, and objectives where it appears. The variable is removed from the `SOS1`, `SOS2`, and `IloConversion` where it appears.

Example

This example tests the linear mode deletion of a variable `x`.

```
void TestLinearDeleter() {
  IloEnv env;
  env.out() << "TestLinearDeleter" << endl;
  try {
    IloModel model(env);
    IloNumVar x(env, 0, 10, "x");
    IloNumVar y(env, 0, 10, "y");
    IloConstraint con = (x + y <= 0);
    IloConstraint con2 = y >= 6;
    IloNumVarArray ar(env, 2, x, y);
    IloSOS1 sos(env, ar, "sos");
    model.add(con);
    model.add(con2);
    model.add(sos);
    env.out() << "Before Delete" << endl;
    env.out() << model << endl;
    x.end();
    con2.end();
    env.out() << "After Delete" << endl;
    env.out() << model << endl;
  } catch (IloException& e) {
    cout << "Error : " << e << endl;
  }
  env.end();
}
```

The example produces the following output:

```
TestLinearDeleter
Before Delete
IloModel model0 = {
IloRange rng3(    1 * x + 1 * y ) <= 0

IloRange rng46 <=(    1 * y )

IloSOS1I (sos)
  _varArray [x(F)[0..10], y(F)[0..10]]
  _valArray []

}

After Delete
IloModel model0 = {
IloRange rng3(    1 * y ) <= 0

IloSOS1I (sos)
  _varArray [y(F)[0..10]]
  _valArray []
}
```

Safe Generic Mode

To use safe generic mode, you must:

- call `IloEnv::setDeleter(IloSafeDeleterMode)`, and
- add `#include <ilconcert/ilodeleter.h>` to your program.

In this mode, the environment builds a dependency graph between all extractables. This graph contains all extractables created

- after a call to `IloEnv::setDeleter(IloSafeDeleterMode)` and
- before a call to `IloEnv::unsetDeleter()`.

Objects not managed by this dependency graph are referred to here as "nondeletable". An attempt to delete a nondeletable object will throw an exception.

We recommended that you create this graph just after the creation of the environment and that you refrain from using `IloEnv::unsetDeleter`. We make these recommendations because building an incomplete dependency graph is very error prone and should only be attempted by advanced users. A good example of this incomplete graph is the separation of a model between a nondeletable base model and deletable extensions of this base model.

Calling `IloExtractable::end()` on extractable `xi` will succeed only if no other extractable uses extractable `xi`. If this is not the case, a call to `IloExtractable::end()` will throw an exception `IloDeleter::RequiresAnotherDeletionException` indicating which extractable uses the extractable that you want to delete.

Example

This example shows an attempt to delete one extractable that is used by another.

```
void TestSafeDeleter() {
  IloEnv env;
  env.out() << "TestSafeDeleter" << endl;
  env.setDeleter(IloSafeDeleterMode);
  try {
    IloModel model(env);
    IloNumVar x(env, 0, 10);
    IloNumVar y(env, 0, 10);
    IloConstraint con = (x + y <= 0);
    try {
      x.end();
    } catch (IloDeleter::RequiresAnotherDeletionException &e) {
      cout << "Caught " << e << endl;
      e.getUsers()[0].end();
      e.end();
    }
    x.end();
  } catch (IloException& e) {
    cout << "Error : " << e << endl;
  }
  env.unsetDeleter();
  env.end();
}
```

The example produces the following output:

```
TestSafeDeleter
Caught You cannot end x1(F)[0..10] before IloRange rng3(  1 * x1 + 1 * x2  ) <= 0
```

To address this, you should use the `IloExtractableArray::endElements()` method. With this method, all extractables appearing in the array are deleted one after another. Thus, if an extractable is used by another extractable and this other extractable is deleted before the first one, the system will not complain and will not throw an exception.

Example

This example illustrates the use of the `endElements()` method

```
void TestSafeDeleterWithArray() {
  IloEnv env;
  env.out() << "TestSafeDeleterWithArray" << endl;
```

```
  env.setDeleter(IloSafeDeleterMode);
  try {
    IloModel model(env);
    IloNumVar x(env, 0, 10);
    IloNumVar y(env, 0, 10);
    IloConstraint con = (x + y <= 0);
    IloExtractableArray ar(env, 2, con, x);
    ar.endElements();
  } catch (IloException& e) {
    cout << "Error : " << e << endl;
  }
  env.unsetDeleter();
  env.end();
}
```

The example will not throw an exception.

Note

Please note that in this last example, the constraint `con` must appear before the variable `x` as it will be deleted before the variable `x`.

Obsolete Functions & Classes

Obsolete Function or Class	Replaced By
`IloNumSet`	`IloIntSet`
`IloNumSetVar`	`IloIntSetVar`
`IloNumSetVarArray`	`IloIntSetVarArray`
`IloExprBase`	`IloNumExprArg` and `IloIntExprArg`
`IloExprNode`	`IloNumExprArg` and `IloIntExprArg`
`IloExprArg`	`IloNumExprArg` and `IloIntExprArg`
`IloExprI`	`IloNumLinTermI` (undocumented)

Assert and NDEBUG - Solver

Most member functions of handle classes in Solver are inline functions that contain an `assert` statement. This statement checks that the handle pointer is non-null. These statements can be suppressed by the macro `NDEBUG`. This option usually reduces execution time. The price you pay for this choice is that attempts to access through null pointers are not trapped and usually result in memory faults.

Choice Point

The ideas of success and failure are used to express algorithms with choices. Indeed, a goal can be defined as a choice between different goals. Such a goal is called a *choice point*.

A choice point is created by the execution of the goal `IlcOr`. A choice point can be labeled when it is created so that you can direct Solver to return to it explicitly.

Here's how a choice point is executed in a depth first search:

- The state of Solver is saved, including the state of the goal stack.
- The first subgoal is added to the top of the goal stack.
- The other subgoals are saved as untried subgoals for the choice point.
- Then the first subgoal is popped from the goal stack and executed. If this subgoal fails, the state of Solver is restored, and the first untried choice is pushed onto the goal stack. This activity is called *backtracking*. Backtracking is done as long as no subgoal succeeds. If no subgoal succeeds, the choice point fails.

Thus a Solver user can define an algorithm without knowing in advance which subgoal will succeed. This kind of programming is often called *non-deterministic* programming.

See the concept Goal for more information.

See Also

IlcAnd, IlcGoal, IlcGoalI, IlcOr

Constraints-Predefined

For the basic classes of Solver, there are predefined constraints ready to use in your applications. This table offers easy access to predefined constraints for basic Solver classes. Complete documentation of each class and each predefined constraint appears in alphabetic order by class name and by function name in this reference manual.

For element constraints, see the function `IlcTableConstraint`.

For this class:	These constraints are predefined:
IlcAnyVarArray	
	IlcAllDiff
	IlcDistribute
	IlcTableConstraint
IlcAnySetVar	
	IlcEqIntersection
	IlcEqPartition
	IlcEqUnion for sets of integer or enumerated expressions
	IlcEqUnion for sets of integer or enumerated values
	IlcEqUnion for sets of sets
	IlcEqUnion for sets
	IlcMember of a set
	IlcNotMember for sets
	IlcNullIntersect
	IlcSubset
	IlcSubsetEq
	IlcUnion for sets of integer or enumerated expressions
	IlcUnion for sets of integer or enumerated values
	IlcUnion for sets of sets

	IlcUnion for sets
IlcAnySetVarArray	
	IlcAllNullIntersect
	IlcEqIntersection
	IlcEqPartition
	IlcEqUnion for sets
	IlcPartition
	IlcUnion for sets
IlcFloatExp	
	IlcNull
IlcFloatVar	
	IlcTableConstraint
IlcIntExp	
	IlcLeOffset
IlcIntArray	
	IlcMember of an array
IlcIntVar	
	IlcTableConstraint
	IlcMinDistance
IlcIntVarArray	
	IlcAllDiff
	IlcAllMinDistance
	IlcDistribute
	IlcInverse
	IlcPathLength
	IlcSequence
	IlcTableConstraint
IlcIntSetVar	
	IlcEqIntersection
	IlcEqPartition
	IlcEqUnion for sets of integer or enumerated expressions
	IlcEqUnion for sets of integer or enumerated values
	IlcEqUnion for sets of sets
	IlcEqUnion for sets
	IlcMember of a set
	IlcNotMember for sets

	IlcNullIntersect
	IlcSetOf
	IlcSubset
	IlcSubsetEq
	IlcUnion for sets of integer or enumerated expressions
	IlcUnion for sets of integer or enumerated values
	IlcUnion for sets of sets
	IlcUnion for sets
IlcIntSetVarArray	
	IlcAllNullIntersect
	IlcEqIntersection
	IlcEqPartition
	IlcEqUnion for sets
	IlcPartition
	IlcUnion for sets

Domain-Delta

The *domain-delta* is a special set where the modifications of the domain of a constrained variable are stored. This domain-delta can be accessed (by means of member functions of the class of the constrained variable) during the propagation of the constraints posted on that constrained variable. When all the constraints posted on that constrained variable have been processed, then the domain-delta is cleared. If the variable is modified again, then the whole process starts over again. The state of the domain-delta is reversible.

The domain-delta can be traversed by an iterator, an instance of `IlcAnyVarDeltaIterator`, `IlcFloatVarDeltaIterator`, or `IlcIntVarDeltaIterator`.

Delta Sets of Constrained Set Variables

When a propagation event is triggered for a constrained set variable (that is, an instance of `IlcIntSetVar` or `IlcAnySetVar`), the constrained set variable is pushed into the constraint propagation queue if it was not already in the queue. Moreover, modifications of the domain of the constrained set variable are stored in two special sets, the *delta sets* (analogous to the domain-delta of a constrained integer variable).

One delta set stores the values removed from the possible set of the constrained set variable, and it is known as the *possible-delta set*. The other delta set stores the values added to the required set of the constrained set variable, and it is known as the *required-delta set*.

Solver provides iterators to traverse the delta sets of a constrained set variable. Those iterators are instances of the classes `IlcIntDeltaPossibleIterator`, `IlcIntDeltaRequiredIterator`, `IlcAnyDeltaPossibleIterator`, or `IlcAnyDeltaRequiredIterator`.

When all the constraints posted on that constrained set variable have been processed, then the delta sets are cleared. If the variable is modified again, then the whole process starts over again. The state of the delta sets is reversible.

The order in which elements of the delta sets are traversed is not predictable. Each newly removed element will be traversed only once.

See the concept Propagation for more information.

See Also

IlcAnyDeltaPossibleIterator , IlcAnyDeltaRequiredIterator, IlcAnySetVar, IlcAnyVarDeltaIterator, IlcFloatVarDeltaIterator, IlcIntDeltaPossibleIterator, IlcIntDeltaRequiredIterator, IlcIntSetVar, IlcIntVarDeltaIterator

Failure

The concept of failure in Solver changed between version 4.4 and version 5.x. In version 4.4, a failure occurred whenever the member function `IlcManager::fail` executed.

In version 5.x, the member functions `IlcConstraintI::fail` and `IlcGoalI::fail` as well as the function `IloGoalFail` cause failure.

Global Modifiers

Solver offers predefined means of controlling its overall behavior. These means are known as global modifiers because they modify globally the behavior you can expect from Solver in your application. You can use global modifiers only in the member function IloSolver::solve.

- One category of global modifiers includes the search limits:
  - IloSolver::setFailLimit
  - IloSolver::setOrLimit
  - IloSolver::setTimeLimit

  Those member functions are implemented by search objects. These search objects can be used with goals:

  - IloFailLimit
  - IloOrLimit
  - IloTimeLimit
- Another category of global modifiers includes the objective minimization step:
  - IloSolver::setOptimizationStep
  - The last parameter of the function IloMinimizeVar

Generic Constraint

A *generic constraint* is a constraint shared by an array of variables. For example, IlcAllDiff is a generic constraint that insures that all the elements of a given array are different from one another. Solver provides generic constraints to save memory since, if you use them, you can avoid allocating one object per variable.

The array makes it easier to implement generic constraints. In fact, member functions of Solver array classes are available to post such generic constraints. A generic constraint is then allocated and recorded only once for all the variables in a given array. This fact represents a significant economy in memory, compared to allocating and recording one constraint per variable.

You create a generic constraint simply by stating the constraint over *generic variables*. Each generic variable stands for all the elements of an array of constrained variables. In order to create generic variables for an array of constrained expressions, Solver provides the class IlcIndex.

See Also

IlcCard, IlcIndex

Goal

Goals are the building blocks used to implement search algorithms in Solver. Both predefined search algorithms and user-defined search algorithms can be expressed in Solver through goals.

Like other Solver entities, a goal is implemented by two objects: a handle (an instance of the class IlcGoal) that contains a data member (the handle pointer) that points to an implementation object (an instance of the class IlcGoalI allocated on the Solver heap).

Among other member functions, the class IlcGoalI has a virtual member function, `execute`, without arguments, which implements the execution of the goal. The `execute` member function must return another goal: the *subgoal* of the goal under execution. If the `execute` member function returns 0 (zero), then no subgoal has to be executed.

A goal can either succeed or fail. A goal fails if a `fail` member function (such as IlcGoalI::fail, for example) is called during its execution. A goal succeeds if it does not fail.

Goal execution is controlled by the member function IloSolver::next and implemented by a goal stack. The first time this member function is called, it pushes all the goals that have been added to the invoking solver onto the goal stack. Then it pops the top of the stack, and if there is a goal there, it executes that goal. When the execution of the current goal is complete, its subgoal is executed (if the current goal has any subgoals). If there are no remaining subgoals, then the next goal on top of the stack is popped. The member function `next` stops when the goal stack is empty.

See the concept Choice Point for more information.

See Also

ILCGOAL0, IlcGoal, IlcGoalI

Goals Predefined

For the basic classes of Solver, there are predefined goals ready to use in your applications. This table offers easy access to predefined goals for basic Solver classes. Complete documentation of each class and each predefined goal is found in this reference manual.

For this class:	These goals are predefined:
`IlcAnySetVar`	
	`IlcBestInstantiate`
	`IlcInstantiate`
`IlcAnySetVarArray`	
	`IlcBestGenerate`
	`IlcGenerate`
`IlcAnyVar`	
	`IlcBestInstantiate`
	`IlcInstantiate`
`IlcAnyVarArray`	
	`IlcBestGenerate`
	`IlcGenerate`
`IlcBoolVarArray`	
	`IlcGenerate`
`IlcFloatVar`	
	`IlcInstantiate`

	IlcBestInstantiate
	IlcGenerateBounds
	IlcSetMax
	IlcSetMin
	IlcSetValue
IlcFloatVarArray	
	IlcBestGenerate
	IlcGenerate
	IlcGenerateBounds
	IlcSplit
IlcGoal	
	IlcAnd
	IlcOr
IlcIntSetVar	
	IlcBestInstantiate
	IlcInstantiate
IlcIntSetVarArray	
	IlcBestGenerate
	IlcGenerate
IlcIntVar	
	IlcBestInstantiate
	IlcInstantiate
	IlcRemoveValue
	IlcSetMax
	IlcSetMin
	IlcSetValue
	IlcDichotomize
IlcIntVarArray	
	IlcBestGenerate
	IlcGenerate
IloAnySetVar	
	IloBestInstantiate
	IloInstantiate
IloAnySetVarArray	
	IloBestGenerate
	IloGenerate

IloAnyVar	
	IloBestInstantiate
	IloInstantiate
IloAnyVarArray	
	IloBestGenerate
	IloGenerate
IloNumSetVar	
	IloBestInstantiate
	IloInstantiate
IloNumSetVarArray	
	IloBestGenerate
	IloGenerate
IloNumVar	
	IloGenerateBounds
	IloInstantiate
	IloBestInstantiate
	IloRemoveValue (for type Int only
	IloSetMax
	IloSetMin
	IloSetValue
	IloDichotomize
IloNumVarArray	
	IloBestGenerate
	IloGenerate
	IloGenerateBounds
	IloSplit
	IloDichotomize

Handle Class - Solver

Most Solver entities are implemented by means of two classes: a handle class and an implementation class, where an object of the handle class contains a data member (the handle pointer) that points to an object (its implementation object) of the corresponding implementation class. As a Solver user, you will be working primarily with handles.

As handles, these objects should be *passed by value*, and they should be created as automatic objects, where “automatic” has the usual C++ meaning.

The name of the implementation class consists of the name of the corresponding handle class followed by the letter I to indicate implementation. For example, the class `IlcConstraint` is a handle class; the class `IlcConstraintI` is the corresponding implementation class.

Member functions of a handle class correspond to member functions of the same name in the implementation class.

Iterator

An *iterator* is an object that traverses a set of other objects. There is an underlying data structure associated with an iterator. The iterator contains a *traversal state* of this data structure. Besides its constructors and destructors, an iterator generally has member functions to access the element at the current position, to check whether the iterator has passed the end position, and to shift the iterator to the next position.

The container for the iterator to scan (for example, an array of elements) is a Solver handle. Likewise, the data in the container (in our example, an element) is also a Solver handle. In both cases—the container and the data—the handle points to an implementation object. From this point of view, you can see that an iterator semantically resembles a string: the container is like the array that defines a string; the data is like characters in the string; a handle pointing to a null implementation object is like the null pointer that customarily terminates a string.

See Also

IlcAnyDeltaPossibleIterator, IlcAnyDeltaRequiredIterator, IlcAnySetIterator, IlcAnyVarArrayIterator, IlcIntDeltaPossibleIterator, IlcIntDeltaRequiredIterator, IlcIntSetIterator, IlcAnyExpIterator, IlcAnySetVarArrayIterator, IlcAnyVarDeltaIterator, IlcIntExpIterator, IlcIntSetVarArrayIterator, IlcIntVarArrayIterator, IlcIntVarDeltaIterator

Propagation

When you post a constraint, the constraint is used immediately to reduce the domains of the constrained variables that it involves. Solver reduces a domain by removing those values that cannot satisfy the constraint and thus cannot participate in a solution.

Posting a constraint is reversible: the constraint is removed when Solver backtracks to choice points set before that constraint was posted.

If constraint propagation causes a domain to be reduced to a single value, then the constrained variable will be bound to that remaining value.

In addition, when you post a constraint, the constraint is saved so that whenever any of the variables to which it applies is modified, the constraint will be activated, and the modification will be transmitted to the other variables that the constraint involves. This activity is called constraint propagation.

The algorithm used for constraint propagation in Solver is straightforward in principle. Solver maintains a queue of variables, called the constraint propagation queue. When a constrained variable is modified, that variable is put at the end of that queue if it is not already in the queue. As long as there are variables in that queue, the algorithm takes the first variable from the queue. We then say this particular variable is in process.

When a variable is processed, it is first removed from the propagation queue. Then each constraint posted on that variable is examined. For one such constraint, all the variables on which it is posted are in turn examined: their domains are reduced by removing those values that are inconsistent with it. If a variable is already in process, then this domain reduction will be deferred until it is no longer in process. If some of these variables are modified during this activity, they, too, are put into the queue if they are not yet in the queue. The algorithm continues as long as there is a variable in the queue to process. The algorithm automatically reduces domains as necessary and halts in either of two situations: when all domains contain only values consistent with the constraints, or when a domain becomes empty. For performance considerations, it does not carry out all the reductions theoretically possible.

This algorithm has several important properties:

- This algorithm always halts.

- It lets you use constraints (such as arithmetic constraints, for example) on more than two variables at a time.
- It lets you handle problems dynamically; that is, you can solve problems where new constraints can be added during the search for a solution.
- Regardless of the order in which the constraints are considered, the domains will always be the same at the end of the execution of the propagation.

See the concepts Choice Point, Domain-Delta, Goal, Propagation Events, Reversibility, and State for more information.

See Also

IlcConstraint

Propagation Events

The examination of the constraints on a variable is triggered by any modification of that variable. There are several kinds of modifications, depending on the class of variable under consideration. We refer to a propagation event as the modification of a constrained variable. There is a key word associated with each of these propagation events.

There are three propagation events:

- `whenValue` means that a value has been assigned to the constrained variable.
- `whenRange` indicates that at least one of the boundaries (the minimum or the maximum) of the domain has been changed. This event is also generated when the variable is bound (in the sense of “assigned a value”).
- `whenDomain` indicates that the domain of a variable has been modified, either when an element is removed from the domain, or when a boundary is modified, or when the variable is bound.

The propagation events that are possible depend on the type of constrained variable under consideration. The following chart shows the correspondence between events and types of variables.

	whenValue	**whenRange**	**whenDomain**
IlcIntExp	yes	yes	yes
IlcAnyExp	yes		yes
IlcFloatExp	yes	yes	
IlcIntSetVar	yes		yes
IlcAnySetVar	yes		yes

These events are triggered only if the variable is actually modified. For example, attempting to remove a value that is not in the domain triggers no event.

These events are used to control when a constraint should be examined. In fact, a constraint can be associated with a given event for a given variable. For example, when a variable is processed in the constraint propagation algorithm, if the `whenDomain` event is the only triggered event, all the constraints associated with the `whenDomain` event are examined. Any constraints associated with the `whenRange` or the `whenValue` events are not examined. These events are thus used for posting constraints.

See the concepts Choice Point, Propagation, and Reversibility for more information.

See Also

IlcConstraint

Reversibility

A reversible class is one where the data members will be restored automatically by Solver when it backtracks.

Objects that use only reversible classes are called reversible objects.

All Solver objects are reversible objects.

Thus, the state of Solver variables, including their domains and the constraints posted on them, are automatically restored when Solver backtracks.

Functions that use only reversible assignments are called reversible functions.

All Solver functions and member functions are reversible functions unless otherwise documented.

In particular, all the member functions and predefined functions for posting constraints in Solver are reversible. Thus, the state of constrained variables, including their domains and the constraints posted on them, is automatically restored when Solver backtracks. Solver saves the state before the function call.

See the concepts Propagation and State for more information.

See Also

IlcConstraint

Selectors

Selectors are objects which select one *object* from a *container* of objects based on some selection criteria. In particular, selectors are used in Solver search goals to select the decision that will be taken at a search node. For example, a selector could be used to select which variable to explore or to select which value to assign to a given variable at a search node.

Selectors of objects of class `Object` from a container `Container` are instances of the template class `IloSelector<Object,Container>`.

For example, selectors of integer variables from an integer variable array are instances of `IloSelector<IlcIntVar,IlcIntVarArray>`.

A selector implements a function `IloBool IloSelector<Object,Container>::select(Object& selected, Container c)` that returns `IloFalse` if and only if no object could be selected and, otherwise, binds the variable `selected` given as the first argument to the selected object.

When the selector is used by a search goal, the member function `select` is called during the execution of the goal to perform the selection of the next decision.

Selectors can either be created on a Concert environment (`IloEnv`) or on the reversible heap of a solver (`IloSolver`) depending on the signature that is used to construct the selector.

There are two ways to define a selector:

- by hard coding the selection of an object using the `ILOSELECTORi` macros
- by using the predefined selector class `IloBestSelector`

When it is possible, IBM advises you to use the second approach as it is more flexible and modular.

The first way to define a selector is to use the `ILOSELECTORi` macros to define the selection code of a selector. Within the code of the macro, the user specifies which element `Object` of the given container of type `Container` is selected by invoking the `select(Object)` method.

For example, you could define a selector of integer variables from an integer variable array that selects a random variable in the array using the following code. This selector has one data field that stores a random generator.

```
ILOSELECTOR1(RandomVariableSelector,
             IlcIntVar,
             IlcIntVarArray, array,
             IloRandom, random) {
   select(array[random.getInt(array.getSize()-1)]); // Selected object
}
```

This macro defines the following two functions, which return a selector allocated either on a Concert environment (`IloEnv`) or on the reversible heap of a solver (`IloSolver`).

```
IloEnv env;
IloSolver solver = ...;
IloRandom random = ...;
IloSelector<IlcIntVar,IlcIntVarArray> sel1 = RandomVariableSelector(env, random);
IloSelector<IlcIntVar,IlcIntVarArray> sel2 = RandomVariableSelector(solver, random);
```

An instance of a variable can then be selected as follows:

```
IlcIntVarArray array = ...;
IlcIntVar selected;
IlcBool isSelected = sel2.select(selected, array);
```

The second way to define a selector is to use the predefined class `IloBestSelector`, which makes the best selection based on criteria that are supplied by parameters. There are many cases where, in order to select an object from a container, all the objects in the container need to be *visited*, filtered by a *predicate*, and then *compared* so that the best object from this comparison will be the selected one.

Such selectors are created using instances of `IloBestSelector<Object,Container>` which is a subclass of `IloSelector<Object,Container>`. This is a flexible and modular way to build a selector and IBM recommends that, when possible, you use this approach.

The `IloBestSelector` constructor can take up to three parameters:

- a *visitor* that allows you to specify how to visit the objects of an instance of container `Container`
- a *predicate* that allows you to filter objects from the container that cannot be selected
- a *comparator* that allows you to compare the candidate objects that have not been filtered and select the best among them

In addition, *evaluators* that evaluate an object and return an `IloNum` value can be used to build comparators. *Translators*, which translate objects of one type to another type, can be used to create both evaluators and predicates.

The concepts of visitor, predicate, comparator, evaluator, and translator are described in the following sections.

Visitors

A visitor `IloVisitor<class Object,class Container>` is a class that allows you to traverse each of the elements of type `Object` of a container class `Container`. For instance, a visitor can be used to specify how to traverse the set of variables `IloIntVar` of an array of variables `IloIntVarArray`.

Visitors can be created in three ways:

- by using the predefined default visitors in Solver
- by defining your own default visitor using the macro `ILODEFAULTVISITOR`
- by defining a new visitor using the `ILOVISITORi` macros

For a pair of classes `<Object,Container>`, you can define at most one visitor that will be considered as the default visitor for objects of type `Object` in container `Container`. By default, if no visitor is given at the construction of an instance of `IloBestSelector<Object,Container>`, the default visitor for the pair `<Object,Container>` will be used.

The following default visitors are predefined in Solver:

- <IloInt,IloIntArray>
- <IloNum,IloNumArray>
- <IloBool,IloBoolArray>
- <IloIntVar,IloIntVarArray>
- <IloNumVar,IloNumVarArray>
- <IloBoolVar,IloBoolVarArray>
- <IlcIntVar,IlcIntVarArray>
- <IlcFloatVar,IlcFloatVarArray>
- <IlcInt,IlcIntArray>
- <IlcFloat,IlcFloatArray>

These visitors traverse the elements of the arrays by increasing index from 0 until array.getSize()-1.

The macro ILODEFAULTVISITOR allows you to define a new default visitor for your own classes of objects and containers. Within the code of this macro, the function visit(Object) allows you to specify the visited objects.

For example, here is how the default visitor for integer variables in an integer variable array is defined in Solver:

```
ILODEFAULTVISITOR(IloIntVar,IloIntVarArray,array) {
   const IloInt size = array.getSize();
   for (IloInt i=0; i<size; ++i)
      visit(array[i]);
}
```

The ILOVISITORi macros allow you to define a new visitor with i data members.

For instance, the visitor defined in the following sample only visits the n first variables in an array of variables, n being a data field of the visitor:

```
ILOVISITOR1(VisitNFirst,
            IlcIntVar,
            IlcIntVarArray, array,
            IlcInt, n) {
   const IloInt imax = IlcMin(n, array.getSize());
   for (IloInt i=0; i<imax; ++i)
      visit(array[i]);
}
```

An instance of such a visitor to visit the 10 first variables of an array can then be constructed as follows:

```
IloVisitor<IlcIntVar,IlcIntVarArray> visitor = VisitNFirst(solver, 10);
```

Predicates

A predicate on objects of type Object is a class that implements a test on an object and returns an IloBool value. Predicates on objects of type Object are instances of the template class IloPredicate<Object>.

Predicates are used in selectors to filter the set of candidate objects for selection.

The test function of a predicate is the member function IloBool operator()(Object obj, IloAny nu =0). This function returns IloTrue if and only if the object satisfies the predicate. An optional context nu can be passed to the test function in cases where the result of the test depends on some contextual information. When predicates are used in an instance of IloBestSelector<Object,Container>, the selector tests each visited object passing the instance of Container as context to the test function of the predicate.

Predicates can either be created on a Concert environment (IloEnv) or on the reversible heap of a solver (IloSolver) depending on the signature that is used to construct the predicate.

There are two ways to define a predicate:

- by defining a new predicate using the macros ILOPREDICATEi or ILOCTXPREDICATEi
- by composing existing predicates

The first way to create a predicate is to use the `ILOPREDICATEi` macros to define new predicates with `i` data members. The `ILOCTXPREDICATEi` macros define new predicates in the same way as the `ILOPREDICATEi` macros, except that you can add a context.

For example, the following code instantiates a predicate that tests whether or not an integer variable is bound:

```
ILOPREDICATE0(IsBound,
              IlcIntVar, v) {
   return v.isBound();
}
```

The slightly more complex predicate that follows instantiates a predicate that tests whether an instance of `IloIntVar` is bound or not in a solver. The solver is stored as a data field of the predicate.

```
ILOPREDICATE1(IsBoundInSolver,
              IloIntVar, v,
              IloSolver, solver) {
   return solver.isBound(v);
}
```

An instance of such a predicate can be allocated on a Concert environment as follows:

```
IloEnv env;
IloSolver solver1 = ...;
IloPredicate<IloIntVar> p1 = IsBoundInSolver(env, solver1);
```

Testing whether a given variable `v` is bound in solver `solver1` can then be performed as follows:

```
IloIntVar v = ...;
IloBool   vIsBoundInSolver1 = p1(v);
```

The second way to create a predicate is to compose subordinate predicates using the logical operators `!`, `&&`, and `||` and the function `IloIfThenElse`.

For instance, if `pi` are predicates on objects of type `<Object>`, a composite predicate on objects of type `<Object>` can be defined as follows:

```
IloPredicate<Object> cp = IloIfThenElse(p0, p1&&!p2, p3) || p4;
```

Predicate `cp` will be true on an instance of object `Object` if and only if for that instance, `p4` is true or, in case `p0` is true, then `p1` is true and `p2` is false, otherwise (if `p0` is false), `p3` is true.

Predicates on objects of different types cannot be composed together. This will raise an error at compilation time.

Composite predicates, when invoked with a context, will pass this context to their subordinate predicates.

Comparators

A comparator is a class that implements a comparison between two objects. Comparators of objects of type `Object` are instances of the template class `IloComparator<Object>`.

Comparators are used in selectors to compare the candidate objects that have not been filtered and select the best among them.

There are two main comparison functions of a comparator: `operator()` and `isBetterThan`.

The function `operator()` takes two objects, `o1` and `o2`, and by default returns -1, 0, or 1 depending on whether `o1` is respectively better, equal or worse than `o2`. The optional argument `nu` is a context that can be used for the comparison:

```
IloInt operator()(IloObject o1, IloObject o2, IloAny nu = 0) const;
```

The function `isBetterThan` takes two objects, `o1` and `o2`, and returns `IloTrue` if and only if `o1` is preferred to `o2`. The optional argument `nu` is a context that can be used for the comparison:

```
IloBool isBetterThan(IloObject o1, IloObject o2, IloAny nu = 0) const;
```

Comparators can either be created on a Concert environment (`IloEnv`) or on the reversible heap of a solver (`IloSolver`) depending on the signature that is used to construct the comparator.

There are three ways to create comparators:

- by defining a new comparator using the `ILOCOMPARATORi`, `ILOCTXCOMPARATORi`, `ILOCLIKECOMPARATORi`, and `ILOCTXCLIKECOMPARATORi` macros
- by composing subordinate comparators using functions such as `IloComposeLexical` and `IloComposePareto`
- by using an evaluator to create a comparator

The first way to define a new comparator is to use macros to define the comparator. Comparators defined with `ILOCOMPARATORi` define a comparator with `i` data fields that must return a Boolean value equal to `IloTrue` if and only if the comparator's left-hand side is better than its right-hand side. The `ILOCTXCOMPARATORi` macros define new comparators in the same way, except that you can add a context for comparison.

For example, the following comparator compares the values of two `IloNumVar` variables in a solution. The solution is a context of the comparison.

```
ILOCTXCOMPARATOR0(HasValueSmallerThan,
                  IloIntVar, v1, v2,
                  IloSolution, sol) {
   return sol.getValue(v1) < sol.getValue(v2);
}
```

An instance of this comparator can be created as follows:

```
IloEnv env;
IloComparator<IloIntVar> comp = HasValueSmallerThan(env);
```

And it can be invoked as follows:

```
IloIntVar v1 = ...;
IloIntVar v2 = ...;
IloSolution sol = ...;
IloBool v1SmallerThanv2InSol = comp(v1,v2,&sol);
```

Comparators defined with `ILOCLIKECOMPARATORi` and `ILOCTXCLIKECOMPARATORi` are very similar to the ones defined by `ILOCOMPARATORi` and `ILOCTXCOMPARATORi` except for the return type of the comparison. The comparators defined by the `ILOCLIKECOMPARATORi` and `ILOCTXCLIKECOMPARATORi` macros return an integer equal to -1 if the comparator's left-hand side is better than its right-hand side, an integer equal to +1 if the comparator's right-hand side is better than its left-hand side, and an integer equal to 0 in cases where they are not comparable.

The second way to define a comparator is to compose subordinate comparators. The class `IloCompositeComparator` allows you to define a subclass of comparator that is composed of a list of subordinate comparators. New comparators can be added to this list by using the member function `add`.

Two of the most useful types of comparator composition are *lexical* composition and *Pareto* composition.

The following code defines a new comparator of objects of type `Object` as a lexical composition of `n` comparators.

```
IloLexicalComparator<Object> lc = IloComposeLexical(c1,c2,...,cn);
```

The composite comparator `lc` will prefer an object `x` to an object `y` if and only if there exists a comparator `ci` in the list of subordinate comparators such that:

- `ci` prefers `x` to `y` and
- all the previous comparators `cj` in the list `(j<i)` could not express any preference between `x` and `y`

The following code defines a new comparator of objects of type `Object` as a Pareto composition of `n` comparators.

```
IloComparator<Object> pc = IloComposePareto(c1,c2,...,cn);
```

The composite comparator `pc` will prefer an object `x` to an object `y` if and only if

- there is no comparator in the list of subordinate comparators that prefers `y` to `x` and
- there exists at least one comparator in the list of subordinate comparators that prefers `x` to `y`

The third way to define a comparator is very common. The first step is to define an evaluator that evaluates the objects and returns a numeric evaluation and then compare the objects depending on this evaluation by preferring the one with the lowest or greatest evaluation.

Evaluators and evaluator-based comparators are described in the next section.

Evaluators

An evaluator of objects of type `Object` is a class that implements an evaluation of an object and returns an `IloNum` value. Evaluators of objects of type `Object` are instances of the template class `IloEvaluator<Object>`.

The evaluation function of an evaluator is the member function `IloNum operator()(Object obj, IloAny nu =0)`. This function returns an evaluation of the instance of the object given as an argument. An optional context `nu` can be passed to the evaluation function in cases where the result of the evaluation depends on some contextual information.

The member functions `makeLessThanComparator` and `makeGreaterThanComparator` are available on the evaluator base class `IloEvaluator` to build and return a comparator based on the invoking evaluator.

Evaluators can either be created on a Concert environment (`IloEnv`) or on the reversible heap of a solver (`IloSolver`) depending on the signature that is used to construct the evaluator.

There are two ways to define an evaluator:

- by defining a new evaluator using the `ILOEVALUATORi` and `ILOCTXEVALUATORi` macros
- by composing existing subordinate evaluators

The first way to create an evaluator is to use the `ILOEVALUATORi` macros to define new evaluators with `i` data members. The `ILOCTXEVALUATORi` macros define new evaluators in the same way as the `ILOEVALUATORi` macros, except that you can add a context.

For example, the following code instantiates an evaluator that returns the size of a variable's domain:

```
ILOEVALUATOR0(DomainSize,
              IlcIntVar, v) {
   return v.getSize();
}
```

The slightly more complex evaluator in the following sample instantiates an evaluator that returns the value of an `IloIntVar` in a solution. The instance of the solution is passed as a context to the evaluation function.

```
ILOCTXEVALUATOR0(ValueInSolution,
                 IloIntVar, v,
                 IloSolution, sol) {
   return sol.getValue(v);
}
```

An instance of such an evaluator can be allocated on a Concert environment as follows:

```
IloEnv env;
IloEvaluator<IloIntVar> eval = ValueInSolution(env);
```

Evaluating a given variable `v` in the context of a given solution `sol` can then be performed as follows:

```
IloIntVar v = ...;
IloSolution sol = ...;
IloNum value = eval(v,&sol);
```

The second way to define an evaluator is to compose subordinate evaluators using algebraic operators such as +, -, *, /, IloMin, and IloMax and miscellaneous functions such as IloCeil or IloFloor.

For instance, if ei are evaluators on objects of type <Object>, a composite evaluator on objects of type <Object> can be defined as follows:

```
IloEvaluator<Object> ce = e1 * IloMin(e2 + IloFloor(3*e3), -e4)
```

Evaluators on objects of different types cannot be composed together. This will raise an error at compilation time.

Composite evaluators, when invoked with a context, will pass this context to their subordinate evaluators.

Composing evaluators and predicates

Evaluators can also be composed with predicates to build new evaluators. Likewise, predicates can be built as a composition of evaluators.

For example, the function IloIfThenElse allows you to instantiate a conditional evaluator that, depending on the test performed by a predicate p, will return the value of different evaluators (e1 if p is true, e2 otherwise):

```
IloEvaluator<Object> ce = IloIfThenElse(p, e1, e2);
```

Composite predicates can also be built as a composition of evaluators with the operators ==, !=, <, <=, >= and >.

For example, the following predicate will return IloTrue on an instance of Object, if and only if, for this instance, the evaluation of e1 is lower than the evaluation of e2:

```
IloPredicate<Object> p = (e1<e2);
```

Composite evaluators and predicates, when invoked with a context, will pass this context to their subordinate evaluators or predicates.

Translators

The template class IloTranslator<ObjectOut,ObjectIn> allows you to translate any object of type ObjectIn into a unique object of type ObjectOut.

Translators can be used to create both evaluators and predicates.

A translator IloTranslator<ObjectOut,ObjectIn> defines a translation member function that, given an input object o and an optional context, returns an output object that is the translation of the object passed as argument:

```
ObjectOut operator()(ObjectIn o, IloAny nu=0)
```

Translators can be used to define new predicates and evaluators by using the composition operator <<.

Translators can either be created on a Concert environment (IloEnv) or on the reversible heap of a solver (IloSolver) depending on the signature that is used to construct the translator.

A new translator can be defined using the macro ILOTRANSLATOR.

For example, suppose an evaluator is available that returns the size of an instance of the Solver class IlcIntSet as defined in the following sample:

```
ILOEVALUATOR0(SetSize,
              IlcIntSet, set) {
   return set.getSize();
}
```

Now suppose that you want to define an evaluator of an `IlcIntSetVar` that returns the size of its required set. You could write it explicitly as follows:

```
ILOEVALUATOR0(RequiredSetSize,
              IlcIntSetVar, v) {
   return v.getRequiredSet().getSize();
}
```

An alternative is to define an object that automatically translates an instance of `IlcIntSetVar` into its unique required set and then compose this translator with the evaluator of `IlcIntSet` to build a new evaluator of `IlcIntSetVar`.

For instance in the above example, a translator that returns the required set of an `IlcIntSetVar` can be defined as follows:

```
ILOTRANSLATOR(RequiredSetTranslator,
              IlcIntSet,
              IlcIntSetVar, v) {
  return v.getRequiredSet();
}
```

The above translator that returns the required set of an `IlcIntSetVar` can then be composed with the evaluator of `IlcIntSet` to build a new evaluator of `IlcIntSetVar`:

```
IloEvaluator<IlcIntSet> sizeOfSet = SetSize(solver);
IloTranslator<IlcIntSet,IlcIntSetVar> requiredSet = RequiredSetTranslator(solver);
IloEvaluator<IlcIntSetVar> requiredSetSize = sizeOfSet << requiredSet;
```

Note that such composed evaluators and predicates, when invoked with a context, will pass this context to their subordinate evaluators/predicates and translators.

Example of a Selector

To demonstrate the concepts explained in this section, the following code sample gives a full example of a complex selector of `IlcIntSetVar` in an `IlcIntSetVarArray` that:

- will only consider the variables with an even index in the array (this is done with a new visitor)
- will only consider unbound variables with a non-empty required set (this is done with a predicate)
- will select an eligible variable with the smallest difference between the size of the possible set and the size of the required set, using the smallest size of possible set to break ties (this is done with a lexical comparator)

```
ILOVISITOR0(VarWithEvenIndex,
            IlcIntSetVar,
            IlcIntSetVarArray, array) {
   const IloInt size = array.getSize();
   for (IloInt i=0; i<size; i+=2)
      visit(array[i]);
}

ILOPREDICATE0(IsBound,
              IlcIntSetVar, v) {
   return v.isBound();
}

ILOEVALUATOR0(SetSize,
              IlcIntSet, set) {
   return set.getSize();
}

ILOTRANSLATOR(RequiredSetTranslator,
              IlcIntSet,
              IlcIntSetVar, v) {
  return v.getRequiredSet();
}

ILOTRANSLATOR(PossibleSetTranslator,
              IlcIntSet,
              IlcIntSetVar, v) {
```

```
  return v.getPossibleSet();
}

IloSolver s = ...;

IloEvaluator<IlcIntSetVar> sizeReq = SetSize(s) << RequiredSetTranslator(s);
IloEvaluator<IlcIntSetVar> sizePos = SetSize(s) << PossibleSetTranslator(s);

IloSelector<IlcIntSetVar,IlcIntSetVarArray> selector =
  IloBestSelector(VarWithEvenIndex(s),
                  !IsBound(s) && (0 < sizeReq),
                  IloComposeLexical((sizePos-sizeReq).makeLessThanComparator(), sizePos.makeLessThanComparator()));
```

See Also

IloBestSelector, IloComparator, IloCompositeComparator, IloEvaluator, IloLexicographicComparator, IloParetoComparator, IloPredicate, IloSelector, IloTranslator, IloVisitor, ILOCLIKECOMPARATOR0, ILOCOMPARATOR0, ILODEFAULTVISITOR, ILOEVALUATOR0, ILOPREDICATE0, ILOSELECTOR0, ILOTRANSLATOR, ILOVISITOR0, IloCeil, IloComposeLexical, IloComposePareto, IloFloor, IloIfThenElse, IloMax, IloMin.

State

In order to be able to search for solutions, a solver (that is, an instance of IloSolver) must save certain information, including the state of solver objects such as:

- the *values* of all the Solver variables associated with that solver,
- the *domains* of all the Solver variables associated with that solver,
- all the *constraints* that have been *posted* on the variables associated with that solver.

The state of a solver also includes memory usage, object values, any search space already explored, and so on.

States are used during the search for a solution. The most important feature of states is that they can be *restored*.

In other words, during the execution of a Solver program, a state can be created, then execution proceeds, then that previously created state may be restored. In such a case, we say that Solver *backtracks* to the previously created state.

See the concepts Choice Point, Propagation, and Reversibility for more information.

See Also

IlcConstraint

Addition of a soft constraint to the solver

A soft constraint is a constraint which can be violated. IBM® ILOG® Solver provides the user with a mechanism which is able to deal with such constraints.

The mechanism used by IBM ILOG Solver to handle soft constraints

Suppose you would like the constraint `ct` to be a soft constraint.

In other words, the constraint `ct` could be violated without triggering a general failure.

IBM ILOG Solver defines a mechanism that gives you all the modifications that could happen in `ct` if it was defined as a hard constraint.

Soft constraints are implemented by a copy mechanism. Thus, if the soft constraint is violated, no general failure is triggered, and if some variables of the soft constraint are modified then the consequences of these modifications are not studied because it is acceptable to violate the constraint `ct`.

Here is a more precise description of this copy mechanism:

- The variables on which the constraint `ct` is defined are copied. These variables are called the copied variables of `ct`. The variables from which the copies are made are called the original variables.
- A new constraint corresponding to the constraint `ct` is defined on the copied variables of `ct`. This constraint is called the soft constraint of `ct`, and `ct` is called the original constraint
- Each time an original variable of `ct` is modified the corresponding copied variable of `ct` is accordingly modified.
- If the soft constraint is violated no global fail is triggered.

It is also possible to link demons to the soft constraint. There are two types of demons:

- Demons that are called when the soft constraint is violated
- Demons that are called when a copied variable is modified

In addition a soft constraint is linked to a specific variable called the `status` variable. This variable is a 0-1 integer variable, which takes the value 1 if the soft constraint must be satisfied and the value 0 if the soft constraint must be violated.

Note
A soft constraint can be defined only for constraints that involve only IlcIntVar variables.

Soft constraints are managed using the class IlcSoftConstraint

Example

This section presents a simple example of use of the soft constraints new functionality.

This example solve a problem of minimization of the number of violated constraints A set of constraint is considered and a new constraint is defined. This new constraint (IlcManageSoftCtI in the code) creates as many soft constraint as the array of constraint contains elements and use a variable which counts the number of violated constraints (variable _obj in the code). This variable is linked to the violated constraints thanks to a demon. Each time a constraint is violated this demon is called, and it increments the counter of violation.

```
/*
 Here is a simple example using soft constraints
 */

#include <ilsolver/ilosolverint.h>
#include <ilsolver/disjunct.h>

ILOSTLBEGIN

class IlcManageSoftCtI : public IlcConstraintI {
  IlcConstraintArray _cons;
  IlcRevInt _numFails;
  IlcIntVar _obj;
  IlcSoftCtHandler _sh;
public:
  IlcManageSoftCtI(IlcConstraintArray cons, IlcIntVar obj):
    IlcConstraintI(cons.getManager()), _cons(cons), _obj(obj), _sh(cons.getManager(),100){}
  void post();
  void propagate();
  void softCtFailDemon(const IlcInt softCtIndex);
  void softCtVarDemon(const IlcInt softCtIndex);
};

ILCCTDEMON1(mySoftCtVarDemon,IlcManageSoftCtI,softCtVarDemon, IlcInt,i);
ILCCTDEMON1(mySoftCtFailDemon,IlcManageSoftCtI,softCtFailDemon, IlcInt, i);
```

```
void IlcManageSoftCtI::softCtFailDemon(const IlcInt softCtIndex){
    // this demon is called when the soft constraint of index softCtIndex is violated
    // the number of fails is incremented, that is
    // the variable _obj, counting the number of violations, is incremented
    // any other action could be done here
  getManager().out() << "constraint: " << _cons[softCtIndex];
  getManager().out() << "is violated" << endl;
  _numFails.setValue(getManager(),_numFails.getValue() +1);
  _obj.setMin(_numFails.getValue());
}

void IlcManageSoftCtI::softCtVarDemon(const IlcInt softCtIndex){
    // this function is called each time a variable involved in the soft constraint is modified
  IlcSoftConstraint softCt=_sh.getSoftConstraint(softCtIndex);
  const IlcInt cvar=softCt.getCopiedVarInProcess();
  getManager().out() << "the copied variable of the variable: ";
  getManager().out() << _sh.getVar(_sh.getVarOfCopiedVar(cvar));
  getManager().out() << " of the constraint: " << _cons[softCtIndex];
  getManager().out() << " has been modified" << endl;
  getManager().out() << "the copied variable is: ";
  getManager().out() << _sh.getCopiedVar(cvar) << endl;
}

void IlcManageSoftCtI::post(){
}

void IlcManageSoftCtI::propagate(){
    // for each constraint in array _cons, a corresponding soft constraint is defined
    // and the demons are linked to the possible modifications
  const IlcInt numCt=_cons.getSize();
  for(IlcInt i=0;i<numCt;i++){
    // creation of the soft constraint
    IlcSoftConstraint softCt=_sh.createSoftConstraint(_cons[i]);
    // addition of demons
    softCt.whenFail(mySoftCtFailDemon(getManager(),this,i));
    softCt.whenDomainReduction(mySoftCtVarDemon(getManager(),this,i));
    // addition of the soft constraint
    add(softCt);
  }
  _sh.getImpl()->printCvars();
  _sh.getImpl()->printSoftCts();
  _sh.getImpl()->printVars();
}

IlcConstraint IlcManageSoftCt(IlcConstraintArray cons,IlcIntVar obj){
  return new (cons.getManager().getHeap()) IlcManageSoftCtI(cons,obj);
}

ILOCPCONSTRAINTWRAPPER2(IloManageSoftCt, solver, IloConstraintArray, _cons, IloIntVar, _obj){
  use(solver, _cons);
  use(solver, _obj);
  return IlcManageSoftCt(solver.getConstraintArray(_cons),
                         solver.getIntVar(_obj));
}

int main(int argc, char** argv) {
  IloEnv env;
  try {
    IloModel model(env);

        // 3 variables are defined
    IloIntVarArray vars(env,3,0,1);

    vars[0].setName("x0");
    vars[1].setName("x1");
    vars[2].setName("x2");

         // 3 constraints are defined
    IloConstraintArray cons(env,3);
    cons[0] = (vars[0]!=vars[1]);
    cons[1] = (vars[0]!=vars[2]);
    cons[2] = (vars[1]!=vars[2]);

        // an objective is defined
```

```
    IloIntVar obj(env,0,3);

        // in order to define these constraints as soft constraints a new constraint is defined
        // this constraint creates a soft constraint for each constraint of cons array
        // on the other hand, this constraint ensures that the objective corresponds to the number of
        // constraints that are violated
    model.add(IloManageSoftCt(env, cons, obj));

        // we search for a solution minimizing the number of violations
    model.add(IloMinimize(env, obj));

    IloSolver solver(model);
    solver.startNewSearch(IloGenerate(env,vars));
    while(solver.next()){
      solver.out() << "obj = " << solver.getIntVar(obj)
                   << endl << solver.getIntVarArray(vars) << endl << endl;
    }
    solver.endSearch();
 }
 catch (IloException& ex) {
   cout << "Error: " << ex << endl;
 }
  env.end();
  return 0;
}
```

Table Constraints

Table constraints are constraints that are used frequently in constraint applications. There are two broad categories of use:

- when external data defines a relation
- when you want to improve the efficiency of the solving process by modeling a subproblem as a table constraint

External data as constraints

In many constraint applications, it is necessary to process a huge quantity of data. For instance, the features of some products can be described as a relation in a database or in text files.

Consider as an example a bicycle factory that can produce thousands of different models. For each model of bicycle, a relation associates the features of that bicycle such as size, weight, color, price. This information can be used in a constraint programming application that allows a customer to find the bicycle that most closely fits a specification. This application benefits from a table constraint: this constraint takes a relation (a set of k-ary tuples) and an array of k variables and guarantees that the values of the variables correspond to a tuple of the relation.

In the bicycle example, you first define the relation as an IloIntTupleSet:

```
IloIntTupleSet bicycleTable(env, 5);
while (thereIsTupleToBeRead()) {
  IloIntArray aTuple(env,5);
  aTuple = myReadTupleFunction();
  // aTuple[0]= the id of a type of bicycle
  // aTuple[1]= its size
  // aTuple[2]= its weight
  // aTuple[3]= its color
  // aTuple[4]= its price
  bicycleTable.add(aTuple);
}
```

Then define an array of integer variables `x`, and assume `x[0]` represents the id of the bicycle, `x[1]` its size, `x[2]` its weight, `x[3]` its color, and `x[4]` its price:

```
IloIntVarArray x(env, 5);
```

Finally, add a table constraint on `x` that forces the values of `x` to be one of the combinations defined in the tuple set:

```
model.add(IloTableConstraint(env, x, bicycleTable, IloTrue);
```

If you are looking for two bicycles of size 2 and 4 with the same color, add another array of integer variables `y` to represent the second bicycle along with another table constraint (notice that the tuple set is shared between the two table constraints):

```
IloIntVarArray y(env, 5);
model.add(IloTableConstraint(env, y, bicycleTable, IloTrue);
model.add(x[1]== 2);
model.add(y[1]== 4);
model.add(x[3]==y[3]);
```

There are several ways to express a table constraint:

- The tuple set may indicate the combinations of values that satisfy the constraint, as in the above example.
- The tuple set may also indicate the combinations of values that do no satisfy the constraint.
- The combinations of values may be expressed by a predicate.
- The combinations of values may be expressed by an array whose index corresponds to one variable and values to a second variable.

See IloTableConstraint for the complete description of the different table constraints.

Improving efficiency: a table constraint for a subproblem

A modeling trick that may sometimes dramatically reduce the CPU time needed to solve a problem consists in identifying a difficult subproblem, computing all the solutions of the subproblem, and storing them in a table constraint. This approach is not restricted to constraint programming, it is a general approach: facing a difficult problem, it is often easier to solve it by:

1. decomposing the problem into subproblems,
2. solving the different subproblems, and
3. connecting the solutions of the subproblems to produce a solution to the whole problem.

A table constraint forces the values of the variables of the subproblem to be one of the solutions of the subproblem. Thus the connection of the solution of the subproblem with the remainder of the problem is automatically handled. The advantage of this approach is that when searching for a solution of the whole problem, instead of always retrieving the solutions of the subproblem in the search, the table constraint forces values to one of these solutions. In other terms, the work of solving the subproblem is factorized and done only once, before the search, and not several times during the search (which is potentially a huge number of times).

This approach also has a drawback: the solutions of the subproblem must be found first so this must be practical (the solutions should not be too numerous). Tables with several hundreds of thousands of tuples can be handled, but subproblems with billions of solutions cannot be handled in this way. Nevertheless, note that it is possible to set a bound on the number of solutions of the subproblem to precompute and store in the table. In this case, the problem solved is a restriction of the initial problem, but this may be useful in practice.

An example of such an approach is available in the file `dinner.cpp` in the `examples/src` directory.

Obsolete Functions & Classes - Solver

The macro ILSOLVER4 makes code written with Solver 4.0 through 4.4 compatible with Solver 5.x.

Obsolete Function or Class	Use This Instead
`ILCDEMON`	`ILCCTDEMON`

IloConstAnyArray	IloArray
IloConstIntArray	IloArray
IloConstNumArray	IloArray
ILOCPGOAL	ILOCPGOALWRAPPER and ILCGOAL
IlcInitFloat	no longer needed
IlcIsInitDone	no longer needed
IlcManager::add	IloSolver::add
IlcManager::addReversibleAction	IloSolver::addReversibleAction
IlcManager::closeLogFile	See the IBM ILOG Concert Technology API
IlcManager::commit	IloSolver::endSearch
IlcManager::end	IloSolver::end
IlcManager::fail	IloSolver::fail
IlcManager::getDefault Precision	IloSolver::getDefaultPrecision
IlcManager::getErrorReporter	IloException in IBM ILOG Concert Technology API
IlcManager::getHeap	IloSolver::getHeap
IlcManager::getMemoryUsage	IloSolver::getMemoryUsage
IlcManager::getNumberOfChoicePoints	IloSolver::getNumberOfChoicePoints
IlcManager::getNumberOfConstraints	IloSolver::getNumberOfConstraints
IlcManager::getNumberOfFails	IloSolver::getNumberOfFails
IlcManager::getNumberOfVariables	IloSolver::getNumberOfVariables
IlcManager::getStream	See the IBM ILOG Concert Technology API
IlcManager::getTime	IloSolver::getTime
IlcManager::getTraceMode	IlcTrace
IlcManager::IlcManager	IloSolver::IloSolver
IlcManager::nextSolution	IloSolver::next
IlcManager::openLogFile	See the IBM ILOG Concert Technology API
IlcManager::out	IloAlgorithm::out in IBM ILOG Concert Technology API
IlcManager::parseTrace	IlcTrace
IlcManager::printInformation	IloSolver::printInformation
IlcManager::remove	IloModel::remove in IBM ILOG Concert Technology API
IlcManager::restart	IloSolver::restartSearch
IlcManager::setDefaultPrecision	IloSolver::setDefaultPrecision
IlcManager::setErrorReporter	IloException in IBM ILOG Concert Technology API
IlcManager::setFailLimit	IloSolver::setFailLimit

IlcManager::setObjMin	IloSolver::setObjMin
IlcManager::setObjMin(-objective)	IloSolver::setObjMin(-objective)
IlcManager::setOrLimit	IloSolver::setOrLimit
IlcManager::setStream	See the IBM ILOG Concert Technology API
IlcManager::setTimeLimit	IloSolver::setTimeLimit
IlcManager::setTraceHook	IlcTrace
IlcManager::solve	IloSolver::solve
IlcManager::solveAll	IloSolver::solve
IlcSearch::end	IloSolver::endSearch
IlcSearch::next	IloSolver::next
IlcSearch::restart	IloSolver::restartSearch
IlcSearch::setObjMin	IloSolver::setObjMin
IlcSearch::setObjMin(-objective)	IloSolver::setObjMin(-objective)
IlcSearchSelectorI::registerSolution	IlcSearchSelectorI::whenLeaf
IlcSearchSelectorI::update	IlcSearchSelectorI::updateObjective
IlcSearchSelectorI::updateTo	IlcSearchSelectorI::updateObjectiveTo
IlcSearchSelectorI::whenFinished	IlcSearchSelectorI::whenFinishedTree
IloSolver::newSearch	IloSolver::startNewSearch
IlcTraceHook	IlcTrace
IlcTraceHookI	IlcTraceI

Group optim.concert

The IBM® ILOG® Concert API.

Class Summary	
IloAlgorithm	The base class of algorithms in Concert Technology.
IloAlgorithm::CannotExtractException	The class of exceptions thrown if an object cannot be extracted from a model.
IloAlgorithm::CannotRemoveException	The class of exceptions thrown if an object cannot be removed from a model.
IloAlgorithm::Exception	The base class of exceptions thrown by classes derived from IloAlgorithm.
IloAlgorithm::NotExtractedException	The class of exceptions thrown if an extractable object has no value in the current solution of an algorithm.
IloAllDiff	For constraint programming: constrains integer variables to assume different values in a model.
IloAllMinDistance	For constraint programming: constraint on the minimum absolute distance between a pair of variables in an array.
IloAnd	Defines a logical conjunctive-AND among other constraints.
IloArray	A template to create classes of arrays for elements of a given class.
IloBarrier	A system class to synchronize threads at a specified number.
IloBaseEnvMutex	A class to initialize multithreading in an application.
IloBoolArray	The array class of the basic Boolean class for a model.
IloBoolVar	An instance of this class represents a constrained Boolean variable in a Concert Technology model.
IloBoolVarArray	The array class of the Boolean variable class.
IloCondition	Provides synchronization primitives adapted to Concert Technology for use in a parallel application.
IloConstraint	An instance of this class is a constraint in a model.
IloConstraintArray	The array class of constraints for a model.
IloDiff	Constraint that enforces inequality.
IloDistribute	For constraint programming:: a counting constraint in a model.
IloEmptyHandleException	The class of exceptions thrown if an empty handle is passed.
IloEnv	The class of environments for models or algorithms in Concert Technology.
IloEnvironmentMismatch	This exception is thrown if you try to build an object using objects from another environment.
IloException	Base class of Concert Technology exceptions.
IloExpr	An instance of this class represents an expression in a model.
IloExprArray	The array class of the expressions class.
IloExpr::LinearIterator	An iterator over the linear part of an expression.
IloExtractable	Base class of all extractable objects.
IloExtractableArray	An array of extractable objects.
IloExtractableVisitor	The class for inspecting all nodes of an expression.
IloFastMutex	

	Synchronization primitives adapted to the needs of Concert Technology.
IloFunction	For constraint programming: A template for creating a handle class to the implementation class built by the template `IloFunctionI`.
IloIfThen	This class represents a condition constraint.
IloIntArray	The array class of the basic integer class.
IloIntBinaryPredicate	For constraint programming: binary predicates operating on arbitrary objects in a model.
IloIntExpr	The class of integer expressions in Concert Technology.
IloIntExprArg	A class used internally in Concert Technology.
IloIntExprArray	The array class of the integer expressions class.
IloIntSet	An instance of this class offers a convenient way to represent a set of integer values.
IloIntSet::Iterator	This class is an iterator that traverses the elements of a finite set of numeric values.
IloIntSetVar	The class `IloIntSetVar` represents a set of integer values.
IloIntSetVarArray	The array class of the set variable class for integer values.
IloIntTernaryPredicate	For constraint programming: ternary predicates operating on arbitrary objects in a model.
IloIntTupleSet	Ordered set of values represented by an array.
IloIntTupleSetIterator	Class of iterators to traverse enumerated values of a tuple-set.
IloIntVar	An instance of this class represents a constrained integer variable in a Concert Technology model.
IloIntVarArray	The array class of the integer constrained variables class.
IloInverse	For constraint programming: constrains elements of one array to be inverses of another.
IloIterator	A template to create iterators for a class of extractable objects.
IloModel	Class for models.
IloModel::Iterator	Nested class of iterators to traverse the extractable objects in a model.
IloMutexDeadlock	The class of exceptions thrown due to mutex deadlock.
IloMutexNotOwner	The class of exceptions thrown.
IloMutexProblem	Exception.
IloNot	Negation of its argument.
IloNumArray	The array class of the basic floating-point class.
IloNumExpr	The class of numeric expressions in a Concert model.
IloNumExprArg	A class used internally in Concert Technology.
IloNumExprArray	The array class of the numeric expressions class.
IloNumExpr::NonLinearExpression	The class of exceptions thrown if a numeric constant of a nonlinear expression is set or queried.
IloNumToAnySetStepFunction	Represents a step function that associates sets with intervals.
IloNumToAnySetStepFunctionCursor	Allows you to inspect the contents of an `IloNumToAnySetStepFunction`.
IloNumToNumSegmentFunction	Piecewise linear function over a segment.
IloNumToNumSegmentFunctionCursor	Cursor over segments of a piecewise linear function.
IloNumToNumStepFunction	Represents a step function that is defined everywhere on an interval.

IloNumToNumStepFunctionCursor	Allows you to inspect the contents of an instance of IloNumToNumStepFunction.
IloNumVar	An instance of this class represents a numeric variable in a model.
IloNumVarArray	The array class of IloNumVar.
IloObjective	An instance of this class is an objective in a model.
IloOr	Represents a disjunctive constraint.
IloPack	For constraint programming: maintains the load of containers, given weighted, assigned items.
IloRandom	This handle class produces streams of pseudo-random numbers.
IloRange	An instance of this class is a range in a model.
IloRangeArray	The array class of ranges for a model.
IloSemaphore	Provides synchronization primitives.
IloSequence	For constraint programming: a sequence constraint in a model.
IloSolution	Instances of this class store solutions to problems.
IloSolutionIterator	This template class creates a typed iterator over solutions.
IloSolution::Iterator	It allows you to traverse the variables in a solution.
IloSolutionManip	An instance of this class accesses a specific part of a solution.
IloTimer	Represents a timer.

Typedef Summary	
IloAny	For constraint programming: the type for objects as variables in enumerations or sets.
IloBool	Type for Boolean values.
IloInt	Type for signed integers.
IloNum	Type for numeric values as floating-point numbers.
IloSolutionArray	Type definition for arrays of `IloSolution` instances.

Macro Summary	
IloFloatVar	An instance of this class represents a constrained floating-point variable in Concert Technology.
IloFloatVarArray	The array class of IloFloatVar.
IloHalfPi	Half pi.
ILOINTBINARYPREDICATE0	For constraint programming: defines a predicate class.
ILOINTTERNARYPREDICATE0	For constraint programming: defines a predicate class.
IloPi	Pi.
IloQuarterPi	Quarter pi.
ILOSTLBEGIN	Macro for STL.
IloThreeHalfPi	Three half-pi.
IloTwoPi	Two pi.

Enumeration Summary	
IloAlgorithm::Status	An enumeration for the class `IloAlgorithm`.
IloDeleterMode	An enumeration to set the mode of an `IloDeleter`.
IloNumVar::Type	An enumeration for the class IloNumVar.

IloObjective::Sense	Specifies objective as minimization or maximization.

Function Summary	
IloAbs	Returns the absolute value of its argument.
IloAbstraction	For constraint programming: returns a constraint that abstracts the values of one array into the abstract value of another array.
IloAdd	Template to add elements to a model.
IloArcCos	Trigonometric functions.
IloBoolAbstraction	For constraint programming: creates a constraint to abstract an array of Boolean variables.
IloCeil	Returns the least integer value not less than its argument.
IloDisableNANDetection	Disables NaN (Not a number) detection.
IloDiv	Integer division function.
IloEnableNANDetection	Enables NaN (Not a number) detection.
IloEndMT	Ends multithreading.
IloExponent	Returns the exponent of its argument.
IloFloor	Returns the largest integer value not greater than the argument.
IloGetClone	Creates a clone.
IloInitMT	Initializes multithreading.
IloIsNAN	Tests whether a double value is a NaN.
IloLexicographic	Returns a constraint which maintains two arrays to be lexicographically ordered.
IloLog	Returns the natural logarithm of its argument.
IloMax	Returns a numeric value representing the max of numeric values.
IloMaximize	Defines a maximization objective.
IloMember	For constraint programming: creates and returns a constraint forcing `element` to be a member of `setVar`.
IloMin	Returns a numeric value representing the min of numeric values.
IloMinimize	Defines a minimization objective.
IloMonotonicDecreasingNumExpr	For constraint programming: creates a new constrained expression equal to `f(x)`.
IloMonotonicIncreasingNumExpr	For constraint programming: creates a new constrained expression equal to `f(x)`.
IloNotMember	For constraint programming: creates and returns a constraint forcing `expr` not to be a member of `elements`.
IloPiecewiseLinear	Represents a continuous or discontinuous piecewise linear function.
IloPower	Returns the power of its arguments.
IloRound	Computes the nearest integer value to its argument.
IloScalProd	Represents the scalar product.
IloScalProd	Represents the scalar product.
IloScalProd	Represents the scalar product.
IloScalProd	Represents the scalar product.
IloSquare	Returns the square of its argument.
IloSum	Returns a numeric value representing the sum of numeric values.

operator new	Overloaded C++ `new` operator.
operator!	Overloaded C++ operator for negation.
operator!=	Overloaded C++ operator.
operator%	Returns an expression equal to the modulo of its arguments.
operator%	Returns an expression equal to the modulo of its arguments.
operator&&	Overloaded C++ operator for conjunctive constraints.
operator*	Returns an expression equal to the product of its arguments.
operator+	Returns an expression equal to the sum of its arguments.
operator-	Returns an expression equal to the difference of its arguments.
operator/	Returns an expression equal to the quotient of its arguments.
operator<	overloaded C++ operator
operator<<	Overloaded C++ operator.
operator<<	Overloaded C++ operator.
operator<=	
operator<=	overloaded C++ operator
operator==	
operator==	Overloaded C++ operator.
operator>	overloaded C++ operator
operator>=	overloaded C++ operator
operator>>	Overloaded C++ operator redirects input.
operator||	Overloaded C++ operator for a disjunctive constraint.

Variable Summary	
ILO_NO_MEMORY_MANAGER	OS environment variable controls Concert Technology memory manager.
IloInfinity	Largest double-precision floating-point number.
IloIntMax	Largest integer.
IloIntMin	Least integer.

Concert Technology offers a C++ library of classes and functions that enable you to design models of problems for both math programming (including linear programming, mixed integer programming, quadratic programming, and network programming) and constraint programming solutions.

Group optim.concert.extensions

The IBM® ILOG® Concert Extensions Library.

Class Summary	
IloCsvLine	Represents a line in a csv file.
IloCsvReader	Reads a formatted csv file.
IloCsvReader::IloColumnHeaderNotFoundException	Exception thrown for unfound header.
IloCsvReader::IloCsvReaderParameterException	Exception thrown for incorrect arguments in constructor.
IloCsvReader::IloDuplicatedTableException	Exception thrown for tables of same name in csv file.
IloCsvReader::IloFieldNotFoundException	Exception thrown for field not found.
IloCsvReader::IloFileNotFoundException	Exception thrown when file is not found.
IloCsvReader::IloIncorrectCsvReaderUseException	Exception thrown for call to inappropriate csv reader.
IloCsvReader::IloLineNotFoundException	Exception thrown for unfound line.
IloCsvReader::IloTableNotFoundException	Exception thrown for unfound table.
IloCsvReader::LineIterator	Line-iterator for csv readers.
IloCsvReader::TableIterator	Table-iterator of csv readers.
IloCsvTableReader	Reads a csv table with format.
IloCsvTableReader::LineIterator	Line-iterator for csv table readers.
IloIntervalList	Represents a list of nonoverlapping intervals.
IloIntervalListCursor	Inspects the intervals of an interval list.

Function Summary	
IloDifference	Creates and returns the difference between two interval lists.
IloDifference	Creates and returns a function equal to the difference between the functions.
IloIntersection	creates and returns a function equal to the intersection between the functions.
IloMax	Creates and returns a function equal to the maximal value of its argument functions.
IloMin	Creates and returns a function equal to the minimal value of its argument functions.
IloUnion	Represents a function equal to the union of the functions.
IloUnion	Creates and returns the union of two interval lists.
operator*	Creates and returns a function equal to its argument function multiplied by a given factor.
operator+	Creates and returns a function equal the sum of its argument functions.
operator-	Creates and returns a function equal to the difference between its argument functions.
operator<<	Overloaded operator for csv output.
operator==	Returns `IloTrue` for same interval lists.
operator==	overloaded operator.
operator==	Overloaded operator tests equality of numeric functions.

Group optim.concert.solver

The IBM® ILOG® Concert Solver API.

Class Summary	
IloAnyArray	For IBM® ILOG® Solver: array class of the enumerated type definition `IloAny`.
IloAnyBinaryPredicate	For IBM ILOG Solver: defines binary predicates on objects in a model.
IloAnySet::Iterator	For IBM® ILOG® Solver: an iterator to traverse the elements of `IloAnySet`.
IloAnySetVar	For IBM® ILOG® Solver: a class to represent a set of enumerated values as a constrained variable.
IloAnySetVarArray	For IBM® ILOG® Solver: array class of the set variable class `IloAnySetVar`.
IloAnyTernaryPredicate	For IBM ILOG Solver: defines ternary predicates on objects in a model.
IloAnyTupleSet	Ordered set of values as an array.
IloAnyTupleSetIterator	Iterator to traverse enumerated values of a tuple-set.
IloAnyVar	For IBM® ILOG® Solver: a class to represent an enumerated variable.
IloAnyVarArray	For IBM® ILOG® Solver: a class to represent an array of enumerated variables.
IloBox	For IBM ILOG Solver: multidimensional boxes for multidimensional placement problems.
IloPathLength	For IBM® ILOG® Solver: a constraint on accumulations along a path.
IloPathTransitI	For IBM® ILOG® Solver: a transit function in a path constraint.

Typedef Summary	
IloNumFunction	For IBM® ILOG® Solver: the type for a pointer to a numeric function.
IloPathTransitFunction	For IBM® ILOG® Solver: a pointer to a function that computes a transit cost of connecting two nodes.

Macro Summary	
ILOANYBINARYPREDICATE0	For IBM ILOG Solver: defines a binary predicate class.
ILOANYTERNARYPREDICATE0	For IBM® ILOG® Solver: defines a ternary predicate class.
IloFloatArray	`IloFloatArray` is the array class of the basic floating-point class.

Function Summary	
IloAllNullIntersect	For IBM® ILOG® Solver: a constraint forcing one set to have no elements in common with another set.
IloCard	For constraint programming: creates and returns a constrained numeric variable that represents the number of elements in `vars`.
IloEqIntersection	For IBM® ILOG® Solver: a constraint forcing the intersection of two sets to the elements of a third set.
IloEqMax	For IBM® ILOG® Solver: a constraint forcing a variable to the maximum of returned values.
IloEqMin	For IBM® ILOG® Solver: a constraint forcing a variable to the minium of returned values.
IloEqPartition	For IBM® ILOG® Solver: a constraint forcing the value of a variable to be required by one set variable in an array.
IloEqSum	For IBM® ILOG® Solver: a constraint forcing a variable to the sum of returned values.
IloEqUnion	For IBM® ILOG® Solver: a constraint forcing the union of two sets to be the elements of a third set.

IloEqUnion	For IBM® ILOG® Solver : a constraint forcing the union of two sets to be the elements of a third set.
IloNullIntersect	For IBM® ILOG® Solver: a constraint forcing one set to have no elements in common with another set.
IloPartition	For IBM® ILOG® Solver: a constraint forcing each value of an array to be required by one set variable in an array.
IloSubset	For IBM® ILOG® Solver: a constraint forcing one set to be strictly a subset of another set.
IloSubsetEq	For IBM® ILOG® Solver: a constraint forcing one set to be a subset of or equivalent to another set.
IloTableConstraint	For IBM® ILOG® Solver: defines simple constraints that are not predefined.

This group contains IBM ILOG Concert classes and functions specific to IBM ILOG Solver or common to IBM ILOG CP Optimizer and IBM ILOG Solver.

Group optim.concert.xml

The IBM ILOG Concert Serialization API.

Class Summary
IloXmlContext
IloXmlInfo
IloXmlReader
IloXmlWriter

Group optim.solver

The IBM® ILOG® Solver API.

Class Summary	
IlcAnyArray	
IlcAnyDeltaPossibleIterator	
IlcAnyDeltaRequiredIterator	
IlcAnyExp	
IlcAnyExpIterator	
IlcAnyPredicate	
IlcAnyPredicateI	
IlcAnySelect	
IlcAnySet	
IlcAnySetArray	
IlcAnySetIterator	
IlcAnySetSelect	
IlcAnySetVar	
IlcAnySetVarArray	
IlcAnySetVarArrayIterator	
IlcAnyToIntExpFunction	
IlcAnyToIntFunction	
IlcAnyTupleSet	
IlcAnyVar	
IlcAnyVarArray	
IlcAnyVarArrayIterator	
IlcAnyVarDeltaIterator	
IlcBoolVar	
IlcBoolVarArray	
IlcBox	
IlcBoxIterator	
IlcConstAnyArray	
IlcConstFloatArray	
IlcConstIntArray	
IlcConstraint	
IlcConstraintAggregator	
IlcConstraintArray	
IlcConstraintI	
IlcDemon	
IlcDemonI	
IlcFloatArray	
IlcFloatExp	

IlcFloatExpIterator
IlcFloatSet
IlcFloatSetIterator
IlcFloatVar
IlcFloatVarArray
IlcFloatVarDeltaIterator
IlcGoal
IlcGoalI
IlcIndex
IlcIntArray
IlcIntDeltaPossibleIterator
IlcIntDeltaRequiredIterator
IlcIntExp
IlcIntExpIterator
IlcIntPredicate
IlcIntPredicateI
IlcIntSelect
IlcIntSelectEvalI
IlcIntSelectI
IlcIntSet
IlcIntSetArray
IlcIntSetIterator
IlcIntSetSelect
IlcIntSetVar
IlcIntSetVarArray
IlcIntSetVarArrayIterator
IlcIntToFloatExpFunction
IlcIntToFloatExpFunctionI
IlcIntToIntExpFunction
IlcIntToIntExpFunctionI
IlcIntToIntStepFunction
IlcIntToIntStepFunctionCursor
IlcIntTupleSet
IlcIntVar
IlcIntVarArray
IlcIntVarArrayIterator
IlcIntVarDeltaIterator
IlcMTNodeEvaluatorI
IlcMTSearchLimitI
IlcMTSearchSelectorI
IlcMemoryManagerI

IlcNodeEvaluator
IlcNodeEvaluatorI
IlcPathTransit
IlcPathTransitEvalI
IlcPathTransitI
IlcPrintTrace
IlcRandom
IlcRevAny
IlcRevBool
IlcRevFloat
IlcRevInt
IlcSearchLimit
IlcSearchLimitI
IlcSearchMonitor
IlcSearchMonitorI
IlcSearchNode
IlcSearchSelector
IlcSearchSelectorI
IlcSoftConstraint
IlcSoftCtHandler
IlcTrace
IlcTraceI
IloAnySetValueSelector
IloAnySetValueSelectorI
IloAnyValueSelector
IloAnyValueSelectorI
IloBestSelector
IloBranchSelector
IloBranchSelectorI
IloCPConstraintI
IloCPTrace
IloCPTraceI
IloComparator
IloCompositeComparator
IloCustomizableGoal
IloEvaluator
IloExplainer
IloGoal
IloGoalI
IloIntSetValueSelector
IloIntSetValueSelectorI

IloIntValueSelector	
IloIntValueSelectorI	
IloLexicographicComparator	This class composes comparators lexicographically.
IloNodeEvaluator	
IloNodeEvaluatorI	
IloParallelSolver	
IloParetoComparator	This class performs Pareto comparison of objects.
IloPredicate	
IloSearchLimit	
IloSearchLimitI	
IloSearchSelector	
IloSearchSelectorI	
IloSelector	
IloSolver	
IloSolverExplainer	
IloTranslator	
IloVisitor	

Typedef Summary
IlcAny
IlcBool
IlcChooseAnyIndex
IlcChooseAnySetIndex
IlcChooseFloatIndex
IlcChooseIntIndex
IlcChooseIntSetIndex
IlcEvalAny
IlcEvalAnySet
IlcEvalInt
IlcEvalIntSet
IlcFloat
IlcFloatFunction
IlcFloatVarRef
IlcInt
IlcIntVarRef
IlcPathTransitFunction

Macro Summary
ILCANYPREDICATE0
ILCARRAY
ILCARRAY2
IlcChooseAnyIndex1

IlcChooseAnyIndex2
IlcChooseFloatIndex1
IlcChooseFloatIndex2
IlcChooseIndex1
IlcChooseIndex2
ILCCTDEMON0
ILCGOAL0
IlcHalfPi
IlcInfinity
IlcIntMax
IlcIntMin
ILCINTPREDICATE0
IlcPi
IlcQuarterPi
ILCREV
ILCSTLBEGIN
IlcThreeHalfPi
IlcTwoPi
IloChooseIntIndex
ILOCLIKECOMPARATOR0
ILOCOMPARATOR0
ILOCPCONSTRAINTWRAPPER0
ILOCPGOALWRAPPER0
ILOCPTRACEWRAPPER0
ILOEVALUATOR0
ILOPREDICATE0
ILOSELECTOR0
ILOTRANSLATOR
ILOVISITOR0

Enumeration Summary
IlcErrorType
IlcFilterLevel
IlcFilterLevelConstraint
IlcFloatDisplay
IlcSearchMonitorI::IlcPruneMode
IloSolver::FailureStatus
IloSynchronizeMode

Function Summary
ilc_fail_stop_here
ilc_trace_stop_here

IlcAbs
IlcAbstraction
IlcAllDiff
IlcAllDiffAggregator
IlcAllMinDistance
IlcAllNullIntersect
IlcAnd
IlcApply
IlcArcCos
IlcArcSin
IlcArcTan
IlcBestGenerate
IlcBestInstantiate
IlcBestInstantiate
IlcBestInstantiate
IlcBFSEvaluator
IlcBoolAbstraction
IlcBranchImpactVarEvaluator
IlcCard
IlcCard
IlcChooseFirstUnboundAny
IlcChooseFirstUnboundAnySet
IlcChooseFirstUnboundBool
IlcChooseFirstUnboundFloat
IlcChooseFirstUnboundInt
IlcChooseFirstUnboundIntSet
IlcChooseMaxMaxFloat
IlcChooseMaxMaxInt
IlcChooseMaxMinFloat
IlcChooseMaxMinInt
IlcChooseMaxRegretMax
IlcChooseMaxRegretMin
IlcChooseMaxSizeFloat
IlcChooseMaxSizeInt
IlcChooseMinMaxFloat
IlcChooseMinMaxInt
IlcChooseMinMinFloat
IlcChooseMinMinInt
IlcChooseMinRegretMax
IlcChooseMinRegretMin
IlcChooseMinSizeAny

IlcChooseMinSizeAnySet
IlcChooseMinSizeFloat
IlcChooseMinSizeInt
IlcChooseMinSizeIntSet
IlcComputeMax
IlcComputeMin
IlcCos
IlcDegreeInformation
IlcDegreeVarEvaluator
IlcDegToRad
IlcDichotomize
IlcDistribute
IlcElementEq
IlcElementEq
IlcElementNEq
IlcElementNEq
IlcEqAbstraction
IlcEqBoolAbstraction
IlcEqIntersection
IlcEqPartition
IlcEqUnion
IlcEqUnion
IlcEqUnion
IlcEqUnion
IlcExponent
IlcGeLex
IlcGenerate
IlcGenerateBounds
IlcGoalFail
IlcGoalTrue
IlcIfThen
IlcImpactInformation
IlcImpactValueEvaluator
IlcImpactVarEvaluator
IlcInstantiate
IlcInstantiate
IlcInstantiate
IlcIntersection
IlcInverse
IlcInverse
IlcLeLex

IlcLeOffset
IlcLimitSearch
IlcLinearCtAggregator
IlcLocalImpactVarEvaluator
IlcLog
IlcLogicAggregator
IlcMax
IlcMax
IlcMax
IlcMax
IlcMember
IlcMember
IlcMin
IlcMin
IlcMin
IlcMin
IlcMinDistance
IlcMinimizeVar
IlcMonotonicDecreasingFloatExp
IlcMonotonicIncreasingFloatExp
IlcMTBFSEvaluator
IlcMTMinimizeVar
IlcNotMember
IlcNotMember
IlcNull
IlcNullIntersect
IlcOnce
IlcOr
IlcPack
IlcPack
IlcPack
IlcPartition
IlcPathLength
IlcPiecewiseLinear
IlcPower
IlcRadToDeg
IlcRandomValueEvaluator
IlcRandomVarEvaluator
IlcReductionInformation
IlcReductionVarEvaluator
IlcRemoveValue

IlcRestartGoal
IlcScalProd
IlcSelectSearch
IlcSequence
IlcSetMax
IlcSetMin
IlcSetOf
IlcSetValue
IlcSin
IlcSizeOverDegreeVarEvaluator
IlcSizeVarEvaluator
IlcSolveBounds
IlcSplit
IlcSquare
IlcSubset
IlcSubsetEq
IlcSuccessRateValueEvaluator
IlcSuccessRateVarEvaluator
IlcSum
IlcSum
IlcSum
IlcSum
IlcTableConstraint
IlcTan
IlcUnion
IlcUnion
IlcUnion
IlcUnion
IloAddConstraint
IloAndGoal
IloApply
IloApply
IloBestGenerate
IloBestInstantiate
IloBFSEvaluator
IloCeil
IloChooseFirstUnboundInt
IloChooseMaxMaxInt
IloChooseMaxMinInt
IloChooseMaxRegretMax
IloChooseMaxRegretMin

IloChooseMaxSizeInt	
IloChooseMinMaxInt	
IloChooseMinMinInt	
IloChooseMinRegretMax	
IloChooseMinRegretMin	
IloChooseMinSizeInt	
IloComposeLexical	Creates a lexicographic composite comparator from existing comparators.
IloComposePareto	Initializes a Pareto composite comparator from existing comparators.
IloDDSEvaluator	
IloDFSEvaluator	
IloDichotomize	
IloFailLimit	
IloFirstSolution	
IloFloor	
IloGeLex	
IloGenerate	
IloGenerateBounds	
IloGoalFail	
IloGoalTrue	
IloIDFSEvaluator	
IloIfThenElse	Creates a conditional predicate.
IloIfThenElse	Creates a conditional evaluator.
IloInitializeImpactGoal	
IloInstantiate	
IloLeLex	
IloLimitSearch	
IloMax	
IloMaximizeVar	
IloMin	
IloMinimizeVar	
IloOrGoal	
IloOrLimit	
IloRemoveValue	
IloRestartGoal	
IloRestoreSolution	
ILORTTIN	
IloSBSEvaluator	
IloSelectSearch	
IloSetMax	
IloSetMin	
IloSetValue	

IloSolveOnce	
IloSplit	
IloStoreBestSolution	
IloStoreSolution	
IloTimeLimit	
IloUpdateBestSolution	
operator new	
operator!	
operator!	Creates a predicate by negation.
operator!=	
operator!=	
operator!=	Creates a non-equality predicate from two evaluators.
operator&&	
operator&&	Creates a predicate performing AND on two predicates.
operator*	
operator*	
operator+	
operator+	
operator-	
operator-	
operator-	
operator/	
operator/	
operator<	Creates a less-than predicate from two evaluators.
operator<	
operator<	
operator<<	
operator<<	
operator<<	Creates translated predicate.
operator<=	Creates a less-than-or-equal predicate from two evaluators.
operator<=	
operator<=	
operator==	Creates an equality predicate from two evaluators.
operator==	
operator==	
operator>	
operator>	Creates a greater-than predicate from two evaluators.
operator>	
operator>=	
operator>=	Creates a greater-than-or-equal predicate from two evaluators.
operator>=	

operator||	
operator||	Creates a predicate performing OR on two predicates.

Variable Summary
ILC_NO_MEMORY_MANAGER
IlcFloatMax
IlcFloatMin

The IBM® ILOG® Solver API.

Group optim.solver.iim

The IBM® ILOG® Iterative Improvement Methods Optimization Components (IIM) API.

Class Summary	
IlcNeighborIdentifier	
IloEAOperatorFactory	A factory for generating operators over arrays of variables.
IloEvent	Basic event class.
IloExplicitEvaluator	An evaluator whose evaluations are specified explicitly.
IloExplicitEvaluator::Iterator	An iterator which will iterate over all evaluated objects in an explicit evaluator.
IloIIM	Management class for IIM components.
IloLargeNHoodI	Special neighborhood implementation for Large Neighborhood Search.
IloListener	The listener class.
IloMetaHeuristic	
IloMetaHeuristicI	
IloMultipleEvaluator	An explicit evaluator which can be refreshed from a container class.
IloNHood	
IloNHoodArray	
IloNHoodI	
IloNHoodModifierI	
IloNeighborIdentifier	
IloPoolOperator	The pool operator class.
IloPoolOperator::Event	Event produced by pool operators.
IloPoolOperatorFactory	An operator factory class providing services for simplifying operator creation.
IloPoolOperator::InvocationEvent	The event produced by pool operators when they are invoked.
IloPoolOperator::SuccessEvent	The event produced by pool operators when they are involved in the production of a solution.
IloPoolProc	Pool processor.
IloRandomSelector	A selector which chooses objects randomly.
IloRouletteWheelSelector	A selector which chooses objects following a roulette wheel rule.
IloSolutionDeltaCheck	
IloSolutionDeltaCheckI	
IloSolutionPool	A pool of solutions.
IloSolutionPool::AddedEvent	
IloSolutionPool::EndEvent	
IloSolutionPool::Event	
IloSolutionPool::Iterator	
IloSolutionPool::RemovedEvent	
IloTabuSearch	
IloTournamentSelector	A selector which chooses objects following a tournament rule.

Typedef Summary	
IloPoolProcArray	An array of pool processors.
IloSolutionPoolEvaluator	An explicit evaluator of solution pools.
IloSolutionPoolSelector	A selector which selects solutions from pools.

Macro Summary	
ILODECLDEFAULTCOMPARATOR	Macro for declaring a default comparator.
ILODEFAULTCOMPARATOR	Macro for defining a default comparator.
ILODEFINELNSFRAGMENT0	Macro for more easily creating LNS neighborhoods.
ILOIIMLISTENER0	A macro to define custom listeners.
ILOIIMOP0	This macro is used to define operators.
ILOMULTIPLEEVALUATOR0	Defines an evaluator that performs an evaluation of all objects within a container.

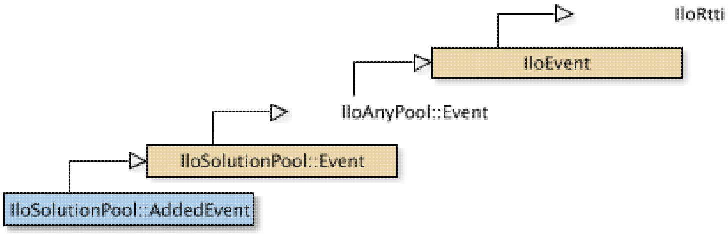
Function Summary	
IloApplyMetaHeuristic	
IloBestSolutionComparator	A solution comparator that prefers higher quality solutions.
IloChangeValue	
IloCommitDelta	
IloCompose	
IloConcatenate	
IloContinue	
IloDestroyAll	Creates a processor which destroys supplied pool elements.
IloExecuteProcessor	Casts a pool processor into an `IloGoal` for execution via an instance of `IloSolver`.
IloFlip	
IloImprove	
IloLimitOperator	Limits the execution of a pool operator.
IloNotify	
IloRandomize	
IloRandomPerturbation	Returns a goal which randomly permutes the search tree branches of another goal.
IloReplaceSolutions	Returns a processor that will replace elements from a supplied pool with elements of the input pool.
IloSample	
IloScanDeltas	
IloScanNHood	
IloSelectProcessor	A pool processor which is a selection from a set of others.
IloSelectSolutions	Creates a pool processor which selects using a standard Solver selector.
IloSetToValue	
IloSimulatedAnnealing	
IloSingleMove	
IloSolutionEvaluator	Evaluates the value of a Boolean variable in a solution.

IloSolutionEvaluator	Evaluates the objective value of a solution.
IloSolutionEvaluator	Evaluates the value of a floating point variable in a solution.
IloSolutionEvaluator	Evaluates the value of an integer variable in a solution.
IloSource	Creates a solution source from a goal and a solution prototype.
IloStart	
IloSwap	
IloTest	
IloTestDelta	
IloUpdate	A goal to update a multiple evaluator.
IloWorstSolutionComparator	A solution comparator that prefers lower quality solutions.
operator&&	Produces the conjunction of two operators.
operator*	
operator+	
operator+	
operator>>	Returns a chain of two pool processors.
operator||	Produces the disjunction of two operators.

The IBM® ILOG® Iterative Improvement Methods Optimization Components (IIM) API.

Class IloSolutionPool::AddedEvent

Definition file: ilsolver/iimpool.h
Include file: <ilsolver/iim.h>



brief Event class used to notify the addition of an `IloSolution` to an `IloSolutionPool`.

This event class is used to notify any listeners that have been attached to the pool using `IloSolutionPool::addListener` whenever an object of type `IloSolution` is added to an `IloSolutionPool` using `IloSolutionPool::add`.

See Also: IloSolutionPool::addListener, IloSolutionPool::add

Method Summary	
`public IloSolution`	`getSolution() const`

Inherited Methods from `Event`
`getPool`

Methods

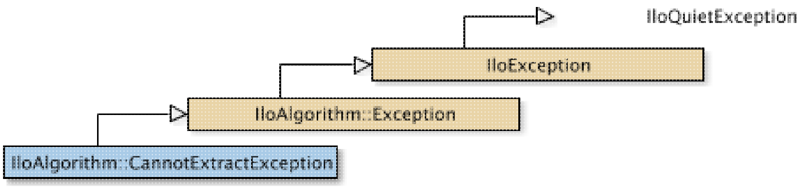
`public IloSolution` **`getSolution`**`() const`

brief Returns the added `IloSolution`.

This function returns the added object of type `IloSolution`.

Class IloAlgorithm::CannotExtractException

Definition file: ilconcert/iloalg.h



The class of exceptions thrown if an object cannot be extracted from a model.
If an attempt to extract an object from a model fails, this exception is thrown.

Method Summary	
public void	end()
public const IloAlgorithmI *	getAlgorithm() const
public IloExtractableArray &	getExtractables()

Inherited Methods from `IloException`
end, getMessage

Methods

`public void` **`end`**`()`

This member function deletes the invoking exception. That is, it frees memory associated with the invoking exception.

`public const IloAlgorithmI *` **`getAlgorithm`**`() const`

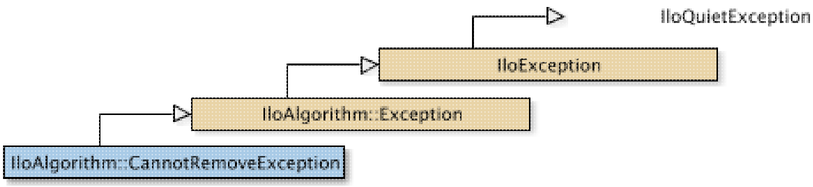
The member function **getAlgorithm** returns the algorithm from which the exception was thrown.

`public IloExtractableArray &` **`getExtractables`**`()`

The member function **getExtractables** returns the extractable objects that triggered the exception.

Class IloAlgorithm::CannotRemoveException

Definition file: ilconcert/iloalg.h



The class of exceptions thrown if an object cannot be removed from a model.
If an attempt to remove an extractable object from a model fails, this exception is thrown.

Method Summary	
public void	end()
public const IloAlgorithmI *	getAlgorithm() const
public IloExtractableArray &	getExtractables()

Inherited Methods from `IloException`
end, getMessage

Methods

`public void` **`end`**`()`

This member function deletes the invoking exception. That is, it frees memory associated with the invoking exception.

`public const IloAlgorithmI *` **`getAlgorithm`**`() const`

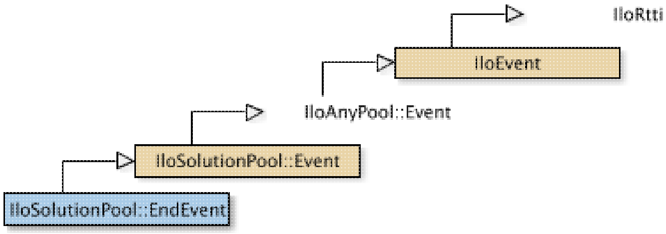
The member function **getAlgorithm** returns the algorithm from which the exception was thrown.

`public IloExtractableArray &` **`getExtractables`**`()`

The member function **getExtractables** returns the extractable objects that triggered the exception.

Class IloSolutionPool::EndEvent

Definition file: ilsolver/iimpool.h
Include file: <ilsolver/iim.h>



brief Event class used to notify the destruction of an `IloSolutionPool`.

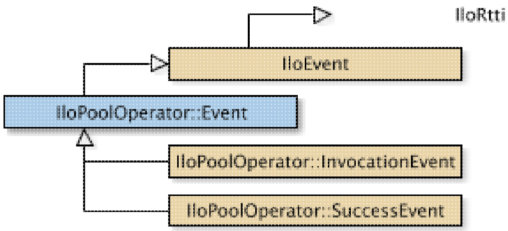
This event class is used to notify any listeners that have been attached to the pool using `IloSolutionPool::addListener` whenever an object of type `IloSolutionPool` is destroyed using `IloSolutionPool:end`.

See Also: IloSolutionPool::addListener, IloSolutionPool::end

Inherited Methods from `Event`
`getPool`

Class IloPoolOperator::Event

Definition file: ilsolver/iimoperator.h
Include file: <ilsolver/iim.h>



Event produced by pool operators.
This class describes events produced by pool operators. Whenever a pool operator is invoked or participates in the creation of a solution (succeeds), it emits an event of this type. These events can be listened to by attaching listeners to operators or to the factories which produce them. Normally, you will not be involved in the *creation* of events, but will only listen to them via listeners; the operators themselves are the creators of events.

See Also: IloPoolOperator::addListener, IloPoolOperatorFactory::addListener, ILOIIMLISTENER0, IloListener

Method Summary	
`public IloPoolOperator`	`getOperator() const` Returns the operator involved.
`public IloSolver`	`getSolver() const` Returns the solver used during execution of the operator.

Methods

`public IloPoolOperator` **`getOperator`**`() const`

Returns the operator involved.

This member function returns the operator `op` that creates the pool operator event.

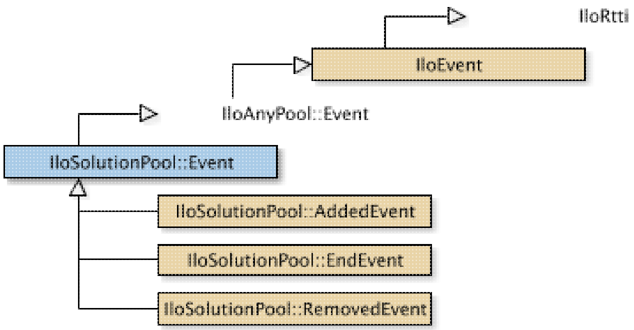
`public IloSolver` **`getSolver`**`() const`

Returns the solver used during execution of the operator.

This member function returns the solver `solver` used during the execution of the operator.

Class IloSolutionPool::Event

Definition file: ilsolver/iimpool.h



brief A general event for `IloSolutionPool` objects which indicates that the pool has changed in some way.

This event class describes a general event for `IloSolutionPool` objects which indicates that the pool has changed in some way. This event notifies any listeners that have been attached to the pool using `IloSolutionPool::addListener`.

See Also: ILOIIMLISTENER0, IloSolutionPool::addListener

Method Summary	
`public IloSolutionPool`	`getPool() const`

Methods

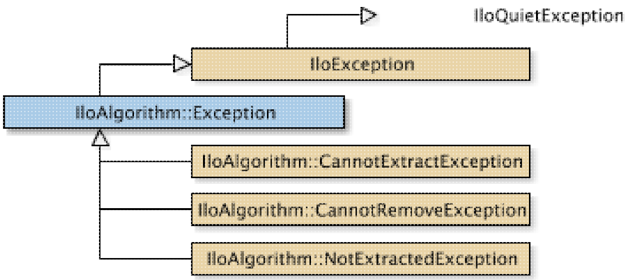
`public IloSolutionPool` **`getPool`**`() const`

brief Returns the pool involved in the event.

This function returns the pool invoked in the event.

Class IloAlgorithm::Exception

Definition file: ilconcert/iloalg.h



The base class of exceptions thrown by classes derived from IloAlgorithm.
IloAlgorithm is the base class of algorithms in Concert Technology.

The class `IloAlgorithm::Exception`, derived from the class IloException, is the base class of exceptions thrown by classes derived from IloAlgorithm.

On platforms that support C++ exceptions, when exceptions are enabled, the member function IloAlgorithm::extract will throw an exception if you attempt to extract an unsuitable object from your model for an algorithm. An extractable object is unsuitable for an algorithm if there is no member function to extract the object from your model to that algorithm.

For example, an attempt to extract more than one objective into an algorithm that accepts only one objective will throw an exception.

Similarly, the member function IloAlgorithm::getValue will throw an exception if you attempt to access the value of a variable that has not yet been bound to a value.

See Also: IloAlgorithm, IloException

Constructor Summary	
`public`	`Exception(const char * str)`

Inherited Methods from `IloException`
`end, getMessage`

Constructors

`public` **`Exception`**`(const char * str)`

This constructor creates an exception thrown from a member of IloAlgorithm. The exception contains the message string str, which can be queried with the member function IloException::getMessage.

Class IlcAnyArray

Definition file: ilsolver/anyexp.h
Include file: <ilsolver/ilosolver.h>

`IlcAnyArray` is the array class for the basic pointer class. It is a handle class.

An object of this class contains a pointer to another object allocated on the Solver heap. Exploiting handles in this way greatly simplifies the programming interface since the handle can then be an automatic object: as a developer using handles, you do not have to worry about memory allocation.

Empty Handle or Null Array

It is possible to create a null array, or in other words, an empty handle. When you do so, only these operations are allowed on that null array:

- **copy**: You can assign the null array to a new array.
- **access its size**: The member function `getSize` for the null array returns 0 (zero).
- **create an iterator**: You can create an iterator to traverse the null array. The member function `ok` returns `IlcFalse` for a null array.

Attempts to access a null array in any other way will throw an exception (an instance of `IloSolver::SolverErrorException`).

See Also: IlcAnyExp, IlcAnySet, IlcConstAnyArray, operator<<

Constructor Summary	
public	IlcAnyArray()
public	IlcAnyArray(IlcInt * impl)
public	IlcAnyArray(IlcAny * impl)
public	IlcAnyArray(IloSolver solver, IlcInt size, IlcAny * values)
public	IlcAnyArray(IloSolver solver, IlcInt size, const IlcAny exp0, const IlcAny exp...)
public	IlcAnyArray(IloSolver solver, IlcInt size)

Method Summary	
public IlcInt *	getImpl() const
public IlcInt	getSize() const
public IloSolver	getSolver() const
public void	operator=(const IlcAnyArray & h)
public IlcAnyExp	operator[](const IlcIntExp rank) const
public IlcAny &	operator[](IlcInt i) const

Constructors

public **IlcAnyArray**()

This constructor creates an empty handle. You must initialize it before you use it.

`public IlcAnyArray(IlcInt * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public IlcAnyArray(IlcAny * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public IlcAnyArray(IloSolver solver, IlcInt size, IlcAny * values)`

This constructor creates an array of pointers containing the values in the array `values`. The argument `size` must be the length of the array `values`; it must also be strictly greater than 0 (zero). Solver does not keep a pointer to the array `values`.

`public IlcAnyArray(IloSolver solver, IlcInt size, const IlcAny exp0, const IlcAny exp...)`

This constructor accepts a variable number of arguments. Its first argument, `size`, indicates the length of the array that this constructor will create; `size` must be the same as the number of instances of `IlcAny` passed as arguments; it must also be strictly greater than 0 (zero). The constructor creates an array of the values indicated by the other arguments (the instances of `IlcAny`).

`public IlcAnyArray(IloSolver solver, IlcInt size)`

This constructor creates an array of `size` elements. The elements of this array are *not* initialized. The argument `size` must be strictly greater than 0 (zero).

Methods

`public IlcInt * getImpl() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public IlcInt getSize() const`

This member function returns the number of elements in the invoking array.

`public IloSolver getSolver() const`

This member function returns an instance of `IloSolver` associated with the invoking object.

`public void operator=(const IlcAnyArray & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

`public IlcAnyExp` **`operator[]`**`(const IlcIntExp rank) const`

This subscripting operator returns a constrained enumerated expression. For clarity, let's call `A` the invoking array. When `rank` is bound to the value `i`, the value of the expression is `A[i]`. More generally, the domain of the expression is the set of values `A[i]` where the `i` are in the domain of `rank`.

`public IlcAny &` **`operator[]`**`(IlcInt i) const`

This operator returns a reference to the element at rank `i`. This operator can be used for accessing the element or for modifying it.

Class IlcAnyDeltaPossibleIterator

Definition file: ilsolver/ilcset.h
Include file: <ilsolver/ilosolver.h>

An instance of the class `IlcAnyDeltaPossibleIterator` is an iterator that traverses the elements of the possible delta set of an instance of `IlcAnySetVar` (a constrained set variable). The order in which the iterator traverses the possible delta set is not predictable.

For more information, see the concepts Propagation, Domain-Delta, and Iterator.

See Also: IlcAnyDeltaRequiredIterator, IlcAnySetVar

Constructor Summary	
public	`IlcAnyDeltaPossibleIterator(IlcAnySetVar var)`

Method Summary	
public IlcBool	`ok() const`
public IlcAny	`operator*() const`
public IlcAnyDeltaPossibleIterator &	`operator++()`

Constructors

`public` **`IlcAnyDeltaPossibleIterator`**`(IlcAnySetVar var)`

This constructor creates an iterator associated with `var` to traverse the values belonging to its possible delta set.

Methods

`public IlcBool` **`ok`**`() const`

This member function returns `IlcTrue` if there is a current element and the iterator points to it. Otherwise, it returns `IlcFalse`.

`public IlcAny` **`operator*`**`() const`

This operator returns the current element, the one to which the invoking iterator points.

`public IlcAnyDeltaPossibleIterator &` **`operator++`**`()`

This operator advances the iterator to point to the next value in the possible delta set.

Class IlcAnyDeltaRequiredIterator

Definition file: ilsolver/ilcset.h
Include file: <ilsolver/ilosolver.h>

An instance of the class `IlcAnyDeltaRequiredIterator` is an iterator that traverses the elements of the required delta set of an instance of `IlcAnySetVar` (a constrained set variable). The order in which the iterator traverses the required delta set is not predictable.

For more information, see the concepts Propagation, Domain-Delta, and Iterator.

See Also: IlcAnyDeltaPossibleIterator, IlcAnySetVar

Constructor Summary	
`public`	`IlcAnyDeltaRequiredIterator(IlcAnySetVar var)`

Method Summary	
`public IlcBool`	`ok() const`
`public IlcAny`	`operator*() const`
`public IlcAnyDeltaRequiredIterator &`	`operator++()`

Constructors

`public` **`IlcAnyDeltaRequiredIterator`**`(IlcAnySetVar var)`

This constructor creates an iterator associated with `var` to traverse the values belonging to its required delta set.

Methods

`public IlcBool` **`ok`**`() const`

This member function returns `IlcTrue` if there is a current element and the iterator points to it. Otherwise, it returns `IlcFalse`.

`public IlcAny` **`operator*`**`() const`

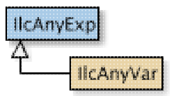
This operator returns the current element, the one to which the invoking iterator points.

`public IlcAnyDeltaRequiredIterator &` **`operator++`**`()`

This operator advances the iterator to point to the next value in the required delta set.

Class IlcAnyExp

Definition file: ilsolver/anyexp.h
Include file: <ilsolver/ilosolver.h>



In order to state constraints on arbitrary objects, Solver defines classes of constrained enumerated expressions and variables. In particular, the class `IlcAnyExp` is the root for constrained enumerated expressions. A constrained enumerated variable is itself a constrained enumerated expression: the class `IlcAnyVar` is a subclass of `IlcAnyExp`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Domain

Each constrained enumerated expression has a domain representing the possible values that can be assigned to the expression. These values are of type `IlcAny`. That is, they are pointers. Those pointers can be pointers to any C++ entity, including objects and instances of user-defined classes.

When the domain contains only one possible value, we say that the expression is *bound*.

The domain of a constrained enumerated expression can be reduced to the point of being empty. In such a case, *failure* is triggered since no solution is then possible.

Implementation Class

The implementation class for `IlcAnyExp` is the class `IlcIntExpI`. In other words, both classes `IlcIntExp` and `IlcAnyExp` have the same implementation class. The member functions for `IlcAnyExp` cast their `IlcAny` arguments into instances of `IlcInt`, then call member functions of `IlcIntExpI`, and cast back the result, if needed.

For example, the member function `setValue` could be defined like this:

```
void IlcAnyExp::setValue(IlcAny value) const {
    assert(getImpl());
    getImpl()->setValue((IlcInt) value);
}
```

Backtracking and Reversibility

All the member functions and operators defined for this class and capable of modifying constrained variables are *reversible*. In particular, the changes made by constraint-posting functions are made with reversible assignments. Thus, the value, the domain, and the constraints posted on any constrained variable are restored when Solver backtracks.

For more information, see the concept Propagation.

A modifier is a member function that reduces the domain of a constrained enumerated expression, if it can. A modifier is not stored, in contrast to a constraint. If the constrained enumerated expression is a constrained enumerated variable, the modifications due to the modifier are stored in its domain. Otherwise, the effect of the modifier is propagated to the subexpressions of the constrained enumerated expression. If the domain becomes empty, a failure occurs. If the domain does not become empty, propagation events are generated for that expression. Modifiers are usually used to define new *classes* of constraints.

See Also: IlcAnyExpIterator, IlcAnySet, IlcAnyVar, IlcAnyVarArray, operator<<

Constructor Summary	
public	IlcAnyExp()
public	IlcAnyExp(IlcIntExpI * impl)

Method Summary	
public IlcAnyExp	getCopy(IloSolver solver) const
public IlcIntExpI *	getImpl() const
public const char *	getName() const
public IlcAny	getObject() const
public IlcInt	getSize() const
public IloSolver	getSolver() const
public IloSolverI *	getSolverI() const
public IlcAny	getValue() const
public IlcBool	isBound() const
public IlcBool	isInDomain(IlcAny value) const
public void	operator=(const IlcAnyExp & h)
public void	removeDomain(IlcAnyArray array)
public void	removeDomain(IlcAnySet set)
public void	removeValue(IlcAny value) const
public void	setDomain(IlcAnyArray array)
public void	setDomain(IlcAnySet set)
public void	setDomain(IlcAnyExp var)
public void	setName(const char * name) const
public void	setObject(IlcAny object) const
public void	setValue(IlcAny value) const
public void	whenDomain(IlcDemon demon) const
public void	whenValue(IlcDemon demon) const

Constructors

public **IlcAnyExp**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IlcAnyExp**(IlcIntExpI * impl)

This constructor creates a handle object from a pointer to an implementation object.

Methods

public IlcAnyExp **getCopy**(IloSolver solver) const

This member function returns a copy of the invoking expression and associates that copy with `solver`.

public IlcIntExpI * **getImpl**() const

This constructor creates an object by copying another one. This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

`public const char * **getName**() const`

This member function returns the name of the invoking object.

`public IlcAny **getObject**() const`

This member function returns a pointer to the external object associated with the invoking object, if there is such an association. It returns 0 (zero) otherwise.

`public IlcInt **getSize**() const`

This member function returns the number of elements in the domain of the invoking expression. In particular, it returns 1 if the invoking constrained enumerated expression is bound.

`public IloSolver **getSolver**() const`

This member function returns an instance of `IloSolver` associated with the invoking object.

`public IloSolverI * **getSolverI**() const`

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

`public IlcAny **getValue**() const`

This member function returns the value of the invoking constrained enumerated expression if that object is bound; otherwise, Solver will throw an exception (an instance of `IloSolver::SolverErrorException`). To avoid errors with `getValue`, you can test expressions by means of `isBound`.

`public IlcBool **isBound**() const`

This member function returns `IlcTrue` if the invoking constrained enumerated expression is bound, that is, if its domain contains only one element. Otherwise, the member function returns `IlcFalse`.

`public IlcBool **isInDomain**(IlcAny value) const`

This member function returns `IlcTrue` if `value` is in the domain of the invoking constrained enumerated expression. Otherwise, the member function returns `IlcFalse`.

`public void **operator=**(const IlcAnyExp & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

`public void **removeDomain**(IlcAnyArray array)`

This member function removes all the elements of the array indicated by `array` from the domain of the invoking constrained expression. If the domain thus becomes empty, then failure occurs. Otherwise, if the domain is modified, the corresponding propagation events are generated. The effects of this member function are reversible.

public void **removeDomain**(IlcAnySet set)

This member function removes all the elements of the set indicated by `set` from the domain of the invoking constrained expression. If the domain thus becomes empty, then failure occurs. Otherwise, if the domain is modified, the corresponding propagation events are generated. The effects of this member function are reversible.

public void **removeValue**(IlcAny value) const

This member function removes `value` from the domain of the invoking constrained expression. If the domain thus becomes empty, then failure occurs. Otherwise, if the domain is modified, the domain propagation event is generated. Moreover, if the invoking constrained enumerated expression becomes bound, then the value propagation event is also generated. The effects of this member function are reversible.

When it removes a value from a domain, Solver may need to allocate more memory for the representation of the remaining domain. The amount of memory allocated depends on the size of the domain of the variable.

public void **setDomain**(IlcAnyArray array)

This member function removes all the elements that are not in the array indicated by `array` from the domain of the invoking constrained expression. If the domain thus becomes empty, then failure occurs. Otherwise, if the domain is modified, the corresponding propagation events are generated. The effects of this member function are reversible.

public void **setDomain**(IlcAnySet set)

This member function removes all the elements that are not in the set indicated by `set` from the domain of the invoking constrained expression. If the domain thus becomes empty, then failure occurs. Otherwise, if the domain is modified, the corresponding propagation events are generated. The effects of this member function are reversible.

public void **setDomain**(IlcAnyExp var)

This member function removes all the elements that are not in domain of `var` from the domain of the invoking constrained expression. If the domain thus becomes empty, then failure occurs. Otherwise, if the domain is modified, the corresponding propagation events are generated. The effects of this member function are reversible.

public void **setName**(const char * name) const

This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

public void **setObject**(IlcAny object) const

This member function establishes a link between the invoking object and an external object of which the invoking object might be a data member.

```
public void setValue(IlcAny value) const
```

This member function removes all the elements that are different from `value` from the domain of the invoking constrained enumerated expression. This has two possible outcomes:

- If `value` was not in the domain of the invoking constrained enumerated expression, the domain becomes empty, and failure occurs.
- If `value` was in the domain, then `value` becomes the value of the expression, and the value and domain propagation events are generated.

The effects of this member function are reversible.

```
public void whenDomain(IlcDemon demon) const
```

This member function associates `demon` with the domain propagation event of the invoking constrained expression. Whenever the domain of the invoking constrained expression changes, the demon will be executed immediately.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. Whenever the domain of the invoking constrained expression changes, the constraint will be propagated.

```
public void whenValue(IlcDemon demon) const
```

This member function associates `demon` with the value propagation event of the invoking constrained expression. Whenever the invoking constrained expression becomes bound, the demon will be executed immediately.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. Whenever the invoking constrained expression becomes bound, the constraint will be propagated.

Class IlcAnyExpIterator

Definition file: ilsolver/anyexp.h
Include file: <ilsolver/ilosolver.h>

An instance of the class `IlcAnyExpIterator` traverses the values belonging to the domain of a constrained enumerated expression (instance of `IlcAnyExp` or `IlcAnyVar`).

For more information, see the concept Iterator.

See Also: IlcAnyExp, IlcAnyVar

Constructor and Destructor Summary	
public	IlcAnyExpIterator(IlcAnyExp exp)

Method Summary	
public IlcBool	ok() const
public IlcAny	operator*() const
public IlcAnyExpIterator &	operator++()

Constructors and Destructors

public **IlcAnyExpIterator**(IlcAnyExp exp)

This constructor creates an iterator associated with `exp` to traverse the values belonging to the domain of `exp`.

Methods

public IlcBool **ok**() const

This member function returns `IlcTrue` if there is a current element and the iterator points to it. Otherwise, it returns `IlcFalse`.

To traverse the values belonging to the domain of a costrained enumerated expression, use the following code:

```
IlcAny val;
for (IlcAnyExpIterator iter(exp); iter.ok(); ++iter){
           val = *iter;
               // do something with val
   }
```

public IlcAny **operator***() const

This operator returns the current element, the one to which the invoking iterator points.

public IlcAnyExpIterator & **operator++**()

This operator advances the iterator to point to the next value in the domain of the constrained enumerated expression.

Class IlcAnyPredicate

Definition file: ilsolver/ilcany.h
Include file: <ilsolver/ilosolver.h>

This class makes it possible for you to define predicates operating on arbitrary objects. A predicate is an object with a method (`IlcAnyPredicate::isTrue`) that checks whether or not a property is satisfied by an ordered set of (pointers to) objects. The ordered set is conventionally represented in Solver by an instance of `IlcAnyArray`.

Defining a New Class of Predicates

Predicates, like other Solver objects, depend on two classes: a handle class, `IlcAnyPredicate`, and an implementation class, `IlcAnyPredicateI`, where an object of the handle class contains a data member (the handle pointer) that points to an object (its implementation object) of an instance of `IlcAnyPredicateI` allocated on the Solver heap. As a Solver user, you will be working primarily with handles.

If you define a new class of predicates yourself, you must define its implementation class together with the corresponding virtual member function `isTrue`, as well as a member function that returns an instance of the handle class `IlcAnyPredicate`.

Arity

As a developer, you can use predicates in Solver applications to define your own constraints that have not already been predefined in Solver. In that case, the *arity* of the predicate (that is, the number of constrained variables involved in the predicate, and thus the size of the array that the member function `IlcAnyPredicate::isTrue` must check) must be less than or equal to three.

See Also: IlcAnyArray, ILCANYPREDICATE0, IlcIntPredicate, IlcTableConstraint

Constructor Summary	
`public`	`IlcAnyPredicate()`
`public`	`IlcAnyPredicate(IlcAnyPredicateI * impl)`

Method Summary	
`public IlcAnyPredicateI *`	`getImpl() const`
`public IlcBool`	`isTrue(IlcAnyArray val)`
`public void`	`operator=(const IlcAnyPredicate & h)`

Constructors

`public` **`IlcAnyPredicate`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcAnyPredicate`**`(IlcAnyPredicateI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

Methods

`public IlcAnyPredicateI * getImpl() const`

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

`public IlcBool isTrue(IlcAnyArray val)`

This member function calls the member function `IlcAnyPredicateI::isTrue`.

`public void operator=(const IlcAnyPredicate & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IlcAnyPredicateI

Definition file: ilsolver/ilcany.h
Include file: <ilsolver/ilosolver.h>

`IlcAnyPredicateI` is the implementation class of `IlcAnyPredicate`, which makes it possible for you to define predicates on arbitrary objects in Solver. A *predicate* is an object with a method (`IlcAnyPredicateI::isTrue`) that checks whether or not a property is satisfied by an ordered set of (pointers to) objects. Conventionally in Solver, the ordered set of objects is represented by an instance of `IlcIntArray`.

Defining a New Class of Predicates

Predicates, like other Solver objects, depend on two classes: a handle class, `IlcAnyPredicate`, and an implementation class, `IlcAnyPredicateI`, where an object of the handle class contains a data member (the handle pointer) that points to an object (its implementation object) of an instance of `IlcAnyPredicateI` allocated on the Solver heap. As a Solver user, you will be working primarily with handles.

If you define a new class of predicates yourself, you must define its implementation class together with the corresponding virtual member function `isTrue`, as well as a member function that returns an instance of the handle class `IlcAnyPredicate`.

Arity

As a developer, you can use predicates in Solver applications to define your own constraints that have not already been predefined in Solver. In that case, the *arity* of the predicate (that is, the number of constrained variables involved in the predicate, and thus the size of the array that the member function `IlcAnyPredicateI::isTrue` must check) must be less than or equal to three.

See Also: IlcAnyArray, ILCANYPREDICATE0, IlcIntPredicateI, IlcTableConstraint

Constructor and Destructor Summary	
`public`	`IlcAnyPredicateI()`
`public`	`~IlcAnyPredicateI()`

Method Summary	
`public virtual IlcBool`	`isTrue(IlcAnyArray val)`

Constructors and Destructors

`public` **`IlcAnyPredicateI`**`()`

This constructor creates an implementation object of a predicate. This constructor should *not* be called directly because this is an *abstract* class. This constructor is called automatically in the constructors of its subclasses.

`public` **`~IlcAnyPredicateI`**`()`

As this class is to be subclassed, a virtual destructor is provided.

Methods

```
public virtual IlcBool isTrue(IlcAnyArray val)
```

This member function must be redefined when you derive a new subclass of `IlcAnyPredicateI`. This member function must return `IlcTrue` if the invoking predicate is satisfied by the elements contained in the array `val`. Otherwise, it must return `IlcFalse`.

Class IlcAnySelect

Definition file: ilsolver/ilcany.h
Include file: <ilsolver/ilosolver.h>

Solver lets you control the order in which the values in the domain of a constrained variable are tried during the search for a solution.

This class is the handle class of the object that chooses the value to try when the constrained variable under consideration is a constrained *enumerated* variable (that is, an instance of `IlcAnyVar`).

An object of this handle class uses the virtual member function `IlcIntSelectI::select` from its implementation class to choose a value in the domain of the constrained variable under consideration during the search for a solution.

See the example in `IlcIntSelect` for a model of how to create and use a selector.

See Also: IlcEvalAny, IlcIntSelectI, IlcIntSelectEvalI, IloAnyValueSelector, IloAnyValueSelectorI

Constructor Summary	
`public`	`IlcAnySelect()`
`public`	`IlcAnySelect(IlcIntSelectI * impl)`
`public`	`IlcAnySelect(const IlcAnySelect & selector)`
`public`	`IlcAnySelect(IloSolver s, IlcEvalAny function)`

Method Summary	
`public IlcIntSelectI *`	`getImpl() const`
`public void`	`operator=(const IlcAnySelect & h)`

Constructors

`public` **`IlcAnySelect`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcAnySelect`**`(IlcIntSelectI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public` **`IlcAnySelect`**`(const IlcAnySelect & selector)`

This copy constructor accepts a reference to an implementation object and creates the corresponding handle object.

`public` **`IlcAnySelect`**`(IloSolver s, IlcEvalAny function)`

This constructor creates a new selector from an evaluation function. The implementation object of the newly created handle is an instance of the class `IlcIntSelectEvalI` constructed with the evaluation function indicated by the argument `function`.

Methods

`public IlcIntSelectI * getImpl() const`

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

`public void operator=(const IlcAnySelect & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IlcAnySet

Definition file: ilsolver/anyexp.h
Include file: <ilsolver/ilosolver.h>

Finite sets of pointers are instances of the handle class `IlcAnySet`. These sets are used by Solver to represent the *domains* of enumerated constrained variables and to represent the *values* of constrained set variables. Solver provides an efficient, optimized implementation of finite sets as bit vectors. These finite sets are known formally as instances of the handle classes `IlcAnySet` or `IlcIntSet`, depending on whether their elements are pointers or integers.

The elements of finite sets of type `IlcAnySet` are pointers of type `IlcAny`. In other words, finite sets of pointers are really sets of pointers to objects of any C++ class.

This class is likely to evolve in future releases of Solver in order to comply with the Standard Template Library adopted by the C++ standard committee.

See Also: IlcAnySetIterator, IlcAnySetVar, IlcAnyArray, IlcAnySetArray, operator<<

Constructor Summary	
public	IlcAnySet()
public	IlcAnySet(IlcIntSetI * impl)
public	IlcAnySet(IloSolver solver, const IlcAnyArray array, IlcBool fullSet=IlcTrue)

Method Summary	
public IlcBool	add(IlcAny elt)
public IlcAnySet	copy() const
public IlcIntSetI *	getImpl() const
public IlcInt	getSize() const
public IloSolver	getSolver() const
public IlcBool	isIn(IlcAny elt) const
public void	operator=(const IlcAnySet & h)
public IlcBool	remove(IlcAny elt)

Constructors

`public` **`IlcAnySet`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcAnySet`**`(IlcIntSetI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public` **`IlcAnySet`**`(IloSolver solver, const IlcAnyArray array, IlcBool` `fullSet=IlcTrue)`

This constructor creates a finite set of pointers containing the elements of `array`. If `array` contains multiple copies of a given value, that value will appear only once in the newly created finite set. If the argument `fullset` is equal to `IlcTrue`, the default value, the finite set will initially contain all its possible values. Otherwise, the finite set will initially be empty. In any case, the possible elements of the finite set are exactly those elements in `array`.

Methods

`public IlcBool` **`add`**`(IlcAny elt)`

This member function adds `elt` to the invoking finite set if `elt` is a possible member of that set and if `elt` is not already in that set. When both conditions are met, this member function returns `IlcTrue`. Otherwise, it returns `IlcFalse`. The effects of this member function are reversible.

`public IlcAnySet` **`copy`**`() const`

This member function creates and returns a finite set that contains the same elements as the invoking finite set.

`public IlcIntSetI *` **`getImpl`**`() const`

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

`public IlcInt` **`getSize`**`() const`

This member function returns the size of a finite set. Clearly, this member function is useful for testing whether the invoking finite set is empty or not.

`public IloSolver` **`getSolver`**`() const`

This member function returns an instance of `IloSolver` associated with the invoking object.

`public IlcBool` **`isIn`**`(IlcAny elt) const`

This member function is a predicate that indicates whether or not `elt` is in the invoking finite set. It returns `IlcTrue` if `elt` is in the set; otherwise, it returns `IlcFalse`.

`public void` **`operator=`**`(const IlcAnySet & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

`public IlcBool` **`remove`**`(IlcAny elt)`

This member function removes `elt` from the invoking finite set. This member function returns `IlcFalse` if `elt` was not in that invoking set. Otherwise, it returns `IlcTrue`. The effects of this member function are reversible.

Class IlcAnySetArray

Definition file: ilsolver/ilcset.h
Include file: <ilsolver/ilosolver.h>

An instance of `IlcAnySetArray` represents an array of sets, instances of `IlcAnySet`. Its implementation class is the undocumented class `IlcIntSetArrayI`.

For each basic type, Solver defines a corresponding array class. This array class is a handle class. In other words, an object of this class contains a pointer to another object allocated on the Solver heap. Exploiting handles in this way greatly simplifies the programming interface since the handle can then be an automatic object: as a developer using handles, you do not have to worry about memory allocation.

Empty Handle or Null Array

It is possible to create a null array, or in other words, an empty handle. When you do so, only these operations are allowed on that null array:

- **copy**: You can assign the null array to a new array.
- **access its size**: The member function `getSize` for the null array returns 0 (zero).
- **create an iterator**: You can create an iterator to traverse the null array. The member function `ok` returns `IlcFalse` for a null array.

Attempts to access a null array in any other way will throw an exception (an instance of `IloSolver::SolverErrorException`).

See Also: IlcAnyArray, IlcAnySet

Constructor Summary	
public	IlcAnySetArray()
public	IlcAnySetArray(IlcIntSetArrayI * impl)
public	IlcAnySetArray(IloSolver solver, IlcInt size, IlcAnySet * values)
public	IlcAnySetArray(IloSolver solver, IlcInt size, IlcAnyArray array)
public	IlcAnySetArray(IloSolver solver, IlcInt size, const IlcAnySet exp...)

Method Summary	
public IlcIntSetArrayI *	getImpl() const
public const char *	getName() const
public IlcInt	getSize() const
public IloSolver	getSolver() const
public IloSolverI *	getSolverI() const
public void	operator=(const IlcAnySetArray & h)
public IlcAnySet &	operator[](IlcInt i) const
public void	setName(const char * name) const

Constructors

`public` **`IlcAnySetArray`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcAnySetArray`**`(IlcIntSetArrayI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public` **`IlcAnySetArray`**`(IloSolver solver, IlcInt size, IlcAnySet * values)`

This constructor creates an array of sets; the length of that array is `size`; its elements are initialized with the values indicated by `values`.

`public` **`IlcAnySetArray`**`(IloSolver solver, IlcInt size, IlcAnyArray array)`

This constructor creates an array of sets; the length of that array is `size`; its elements are initialized with the values indicated by `array`.

`public` **`IlcAnySetArray`**`(IloSolver solver, IlcInt size, const IlcAnySet exp...)`

This constructor creates an array of sets; the length of that array is `size`; its elements are initialized with the arguments of type `IlcAnySet`. The number of arguments of type `IlcAnySet` must be the same as `size`.

Methods

`public IlcIntSetArrayI *` **`getImpl`**`() const`

This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

`public const char *` **`getName`**`() const`

This member function returns the name of the invoking object.

`public IlcInt` **`getSize`**`() const`

This member function returns the number of elements in the invoking array.

`public IloSolver` **`getSolver`**`() const`

This member function returns an instance of `IloSolver` associated with the invoking object.

`public IloSolverI *` **`getSolverI`**`() const`

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

`public void` **`operator=`**`(const IlcAnySetArray & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

```
public IlcAnySet & operator[](IlcInt i) const
```

This operator returns a reference to the element at rank `i`. This operator can be used for accessing the element or for modifying it.

```
public void setName(const char * name) const
```

This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

Class IlcAnySetIterator

Definition file: ilsolver/anyexp.h
Include file: <ilsolver/ilosolver.h>

An instance of the class `IlcAnySetIterator` is an iterator that traverses the elements of a finite set of pointers (instance of `IlcAnySet`).

For more information, see the concept Iterator.

See Also: IlcAnySet

Constructor Summary	
public	IlcAnySetIterator(IlcAnySet set)

Method Summary	
public IlcBool	ok() const
public IlcAny	operator*() const
public IlcAnySetIterator &	operator++()

Constructors

public **IlcAnySetIterator**(IlcAnySet set)

This constructor creates an iterator associated with `set` to traverse its elements.

Methods

public IlcBool **ok**() const

This member function returns `IlcTrue` if there is a current element and the invoking iterator points to it. Otherwise, it returns `IlcFalse`.

To traverse the elements of a finite set of pointers, use the following code:

```
IlcAny val;
for(IlcAnySetIterator iter(set); iter.ok(); ++iter){
        val = *iter;
        // do something with val
}
```

public IlcAny **operator***() const

This operator returns the current element, the one to which the invoking iterator points.

public IlcAnySetIterator & **operator++**()

This operator advances the iterator to point to the next value in the set.

Class IlcAnySetSelect

Definition file: ilsolver/ilcset.h
Include file: <ilsolver/ilosolver.h>

Solver lets you control the order in which the values in the domain of a constrained set variable are tried during the search for a solution.

This class is the handle class of the object that chooses the value to try when the constrained set variable under consideration is a constrained *enumerated* set variable (that is, an instance of `IlcAnySetVar`).

An object of this handle class uses the virtual member function `IlcIntSetSelectI::select` from its implementation class to choose a value in the domain of the constrained enumerated set variable under consideration during the search for a solution.

Example

Here is an example of how to create your own set selector.

```
class myAnySetSelect: public IlcIntSetSelectI {
  public:
    myAnySetSelect(){};
    virtual IlcInt select(IlcIntSetVar var);
};
IlcInt myAnySetSelect::select(IlcIntSetVar var) {
  for(IlcIntSetIterator iter(var.getPossibleSet());
      iter.ok();
      ++iter); {
      if (!var.isRequired(*iter)) return *iter;
      }
  return 0;
}
IlcAnySetSelect mySelect(IloSolver s) {
    return new (s.getHeap()) myAnySetSelect();
}
```

Here is how you use it during a search (inside a goal or constraint, for example).

```
IlcInstantiate(var, mySelect(s));
```

See Also: IlcEvalAnySet, IloInstantiate, IlcIntSetSelect, IloAnySetValueSelector, IloAnySetValueSelectorI

Constructor Summary	
public	`IlcAnySetSelect()`
public	`IlcAnySetSelect(IlcIntSetSelectI * impl)`
public	`IlcAnySetSelect(const IlcAnySetSelect & sel)`
public	`IlcAnySetSelect(IloSolver s, IlcEvalAnySet function)`

Method Summary	
`public IlcIntSetSelectI *`	`getImpl() const`
`public void`	`operator=(const IlcAnySetSelect & h)`

Constructors

`public IlcAnySetSelect()`

This constructor creates an empty handle. You must initialize it before you use it.

`public IlcAnySetSelect(IlcIntSetSelectI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public IlcAnySetSelect(const IlcAnySetSelect & sel)`

This copy constructor accepts a reference to an implementation object and creates the corresponding handle object.

`public IlcAnySetSelect(IloSolver s, IlcEvalAnySet function)`

This constructor creates a new enumerated set selector from an evaluation function. The implementation object of the newly created handle is an instance of the class `IlcIntSetSelectEvalI` constructed with the evaluation function indicated by the argument `function`.

Methods

`public IlcIntSetSelectI * getImpl() const`

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

`public void operator=(const IlcAnySetSelect & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IlcAnySetVar

Definition file: ilsolver/ilcset.h
Include file: <ilsolver/ilosolver.h>

A constrained set variable is any instance of the class `IlcAnySetVar` or `IlcIntSetVar`. The value of a variable belonging to the class `IlcAnySetVar` is an instance of the class `IlcAnySet`. The value of a variable belonging to the class `IlcIntSetVar` is an instance of the class `IlcIntSet`. These two classes are handle classes. They both have the same implementation class, `IlcIntSetVarI`.

Domain

The domain associated with a constrained set variable is a *set of sets*. Solver represents this kind of domain by its upper and lower bounds. The upper bound is the union of all the possible values for the variable, that is the union of all the element-sets of the domain. The lower bound is the intersection of all the possible values of the variables, that is the intersection of all the element-sets of the domain.

In other words, the domain of a constrained set variable is represented by two sets:

- the *required set*, that is, the set of those elements that belong to all the possible values of the variable (the lower bound);
- the *possible set*, that is the set of those elements that belong to at least one of the possible values of the variable (the upper bound).

The possible set contains the required set by construction. The value, the possible set, and the required set of a constrained set variable are all instances of the classes `IlcAnySet` or `IlcIntSet`. When a constrained set variable is *bound*, the required elements are the same as the possible ones, and they are the elements of the *value* of the variable.

Delta Sets and Propagation

When a propagation event is triggered for a constrained set variable, the variable is pushed into the constraint propagation queue if it was not already in the queue. Moreover, the modifications of the domain of the constrained set variable are stored in two special sets. The first set stores the values removed from the possible set of the constrained set variable, and it is called the *possible-delta set*. The second one stores the values added to the required set of the constrained set variable, and it is called the *required-delta set*. These delta sets can be accessed during the propagation of the constraints posted on that variable. When all the constraints posted on that variable have been processed, then the delta sets are cleared. If the variable is modified again, then the whole process begins again. The state of the delta sets is reversible.

Failure

The domain of a costrained set variable can be reduced until it is empty, that is, to the point that the required set is not included in the possible set. In such a case, *failure* occurs since at that point, no solution is possible.

Cardinality (Size of a Set)

It is also possible to constrain the *cardinality* of the value of a constrained set variable. A constrained set variable contains a data member that is a constrained integer variable (called the cardinality variable); it represents how many elements are in the value of the set variable. The minimum of the cardinality variable is always greater than or equal to the size of the required set. Its maximum is always less than or equal to the size of the possible set. The functions `IlcCard` (for sets and for indices) access cardinality.

Backtracking and Reversibility

All member functions defined for this class and capable of modifying constrained set variables are *reversible.* In particular, the changes made by constraint-posting functions are made with reversible assignments. Thus, the value, domain, and constraints posted on any constrained set variables are restored when Solver backtracks.

Modifiers for a constrained set variable reduce its domain. They are usually used when you define a new *class* of constraints.

See Also: IlcAnyDeltaPossibleIterator, IlcAnyDeltaRequiredIterator, IlcAnySet, IlcAnySetVarArray, IlcCard, IlcCard, IlcIntersection, IlcUnion, operator<<

Constructor Summary	
public	IlcAnySetVar()
public	IlcAnySetVar(IlcIntSetVarI * impl)
public	IlcAnySetVar(IloSolver solver, IlcAnyArray array, const char * name=0)

Method Summary	
public void	addRequired(IlcAny elt) const
public IlcAnySetVar	getCopy(IloSolver solver) const
public IlcIntSetVarI *	getImpl() const
public const char *	getName() const
public IlcAny	getObject() const
public IlcAnySet	getPossibleSet() const
public IlcAnySet	getRequiredSet() const
public IlcInt	getSize() const
public IloSolver	getSolver() const
public IloSolverI *	getSolverI() const
public IlcAnySet	getValue() const
public IlcBool	isBound() const
public IlcBool	isInDomain(IlcAnySet set) const
public IlcBool	isInProcess() const
public IlcBool	isPossible(IlcAny elt) const
public IlcBool	isRequired(IlcAny elt) const
public void	operator=(const IlcAnySetVar & h)
public void	removePossible(IlcAny elt) const
public void	setDomain(IlcAnySetVar var) const
public void	setName(const char * name) const
public void	setObject(IlcAny object) const
public void	whenDomain(const IlcDemon demon) const
public void	whenValue(const IlcDemon demon) const

Constructors

public **IlcAnySetVar**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IlcAnySetVar**(IlcIntSetVarI * impl)

This constructor creates a handle object from a pointer to an implementation object.

`public` **`IlcAnySetVar`**`(IloSolver solver, IlcAnyArray array, const char * name=0)`

This constructor creates a constrained set of pointers with no required elements; its possible elements are the pointers in `array`, an array of pointers.

Methods

`public void` **`addRequired`**`(IlcAny elt) const`

The way this member function behaves depends on whether its argument `elt` is a member of the required or possible set of the invoking object. If `elt` is already in the required set of the invoking object, then this member function does nothing. If `elt` is in the possible set of the invoking object, but not yet in the required set, then this member function adds `elt` to the required set. If `elt` is not in the possible set of the invoking object, then failure occurs.

`public IlcAnySetVar` **`getCopy`**`(IloSolver solver) const`

This member function returns a copy of the invoking constrained set variable and associates that copy with the solver indicated by `solver`.

`public IlcIntSetVarI *` **`getImpl`**`() const`

This constructor creates an object by copying another one. This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

`public const char *` **`getName`**`() const`

This member function returns the name of the invoking object.

`public IlcAny` **`getObject`**`() const`

This member function returns a pointer to the external object associated with the invoking object, if there is such an association. It returns 0 (zero) otherwise.

`public IlcAnySet` **`getPossibleSet`**`() const`

This member function returns the possible set of the invoking constrained set variable.

`public IlcAnySet` **`getRequiredSet`**`() const`

This member function returns the required set of the invoking constrained set variable.

`public IlcInt` **`getSize`**`() const`

This member function returns one plus the difference between the cardinality of the set of possible elements and the cardinality of the set of required elements. Don't confuse the size of the *domain* of the constrained set variable (returned by this member function) with the cardinality of the set to which the variable is *bound* (that is, the size of the *value* of the variable). The cardinality of the set variable itself is a constrained integer variable returned by the function `IlcCard`.

`public IloSolver getSolver() const`

This member function returns an instance of `IloSolver` associated with the invoking object.

`public IloSolverI * getSolverI() const`

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

`public IlcAnySet getValue() const`

This member function returns the value of the invoking constrained set variable if that variable has been bound; otherwise, it will throw an exception (an instance of `IloSolver::SolverErrorException`).

`public IlcBool isBound() const`

This member function returns `IlcTrue` if the constrained set variable has been bound, that is, if its set of required elements is equal to its set of possible elements. Otherwise, the member function returns `IlcFalse`.

`public IlcBool isInDomain(IlcAnySet set) const`

This member function returns `IlcTrue` if and only if the finite set indicated by `set` satisfies the following conditions:

- The set contains all the required elements of the invoking constrained set variable.
- Each element of the set is a possible element of the invoking constrained set variable.

`public IlcBool isInProcess() const`

This member function returns `IlcTrue` if the invoking constrained set variable is currently being processed by the constraint propagation algorithm. Only one variable can be in process at a time.

`public IlcBool isPossible(IlcAny elt) const`

This member function returns `IlcTrue` if `elt` is a possible element in the invoking constrained set variable. It returns `IlcFalse` otherwise. This member function could be defined as `getPossibleSet().isIn(elt)`.

`public IlcBool isRequired(IlcAny elt) const`

This member function returns `IlcTrue` if `elt` is a required element in the invoking constrained set variable. It returns `IlcFalse` otherwise. This member function could be defined as `getRequiredSet().isIn(elt)`.

`public void operator=(const IlcAnySetVar & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

`public void` **`removePossible`**`(IlcAny elt) const`

The way this member function behaves depends on whether its argument `elt` is a member of the required or possible set of the invoking object. If `elt` is in the required set of the invoking object, then failure occurs. If `elt` is in the possible set of the invoking object, but not in the required set, then this member function removes `elt` from the possible set. If `elt` is not in the possible set, then this member function does nothing.

`public void` **`setDomain`**`(IlcAnySetVar var) const`

This member function reduces the domain of the invoking constrained set variable so that its domain becomes included in the domain of the constrained set variable `var`.

If the invoking variable is already bound, then this member function considers whether its value belongs to the domain of `var`. If its value does *not* belong to the domain of `var`, then failure occurs.

If the invoking variable is not yet bound, then its required set is replaced by its union with the required set of `var`, and its possible set is replaced by its intersection with the possible set of `var`. If the resulting required set is not included in the resulting possible set, then failure occurs. If the resulting required set contains the same elements as the resulting possible set, then the invoking variable is bound to that remaining value. In any case, if the invoking variable is modified, the constraints posted on it are activated.

The effects of this member function are reversible.

`public void` **`setName`**`(const char * name) const`

This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

`public void` **`setObject`**`(IlcAny object) const`

This member function establishes a link between the invoking object and an external object of which the invoking object might be a data member.

`public void` **`whenDomain`**`(const IlcDemon demon) const`

This member function associates `demon` with the domain propagation event of the invoking constrained set variable. Whenever the domain of the invoking constrained set variable changes, the demon will be executed immediately.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. Whenever the domain of the invoking constrained set variable changes, the constraint will be propagated.

`public void` **`whenValue`**`(const IlcDemon demon) const`

This member function associates `demon` with the value propagation event of the invoking constrained set variable. Whenever the invoking constrained set variable becomes bound, the demon will be executed immediately.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. Whenever the invoking constrained set variable becomes bound, the constraint will be propagated.

Class IlcAnySetVarArray

Definition file: ilsolver/ilcset.h
Include file: <ilsolver/ilosolver.h>

An *array* of constrained set variables of pointers is an instance of the class `IlcAnySetVarArray`. The elements of such an array are instances of `IlcAnySetVar`.

Empty Handle or Null Array

It is possible to create a null array, or in other words, an empty handle. When you do so, only these operations are allowed on that null array:

- **copy**: You can assign the null array to a new array.
- **access its size**: The member function `getSize` for the null array returns 0 (zero).
- **create an iterator**: You can create an iterator to traverse the null array. The member function `ok` returns `IlcFalse` for a null array.

Attempts to access a null array in any other way will throw an exception (an instance of `IloSolver::SolverErrorException`).

See Also: IlcAnySetVar, IlcAnySetVarArrayIterator, operator<<

Constructor Summary	
public	IlcAnySetVarArray()
public	IlcAnySetVarArray(IlcIntSetVarArrayI * impl)
public	IlcAnySetVarArray(IloSolver solver, IlcInt size)
public	IlcAnySetVarArray(IloSolver s, IlcInt size, IlcAnyArray array)
public	IlcAnySetVarArray(IloSolver solver, IlcInt size ILCPARAM, const IlcAnySetVar v1, const IlcAnySetVar v2)

Method Summary	
public IlcAnySetVarArray	getCopy(IloSolver solver) const
public IlcIntSetVarArrayI *	getImpl() const
public const char *	getName() const
public IloSolver	getSolver() const
public IloSolverI *	getSolverI() const
public void	operator=(const IlcAnySetVarArray & h)
public IlcAnySetVar &	operator[](IlcInt index) const
public void	setName(const char * name) const

Constructors

`public` **`IlcAnySetVarArray`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcAnySetVarArray`**`(IlcIntSetVarArrayI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public **IlcAnySetVarArray**(IloSolver solver, IlcInt size)`

This constructor creates an array of `size` uninitialized constrained set variables. The index range of the array is `[0 size)`, where 0 is included but `size` is excluded. The argument `size` must be strictly greater than 0 (zero). Each element of the array must be assigned a value before the array can be used.

`public **IlcAnySetVarArray**(IloSolver s, IlcInt size, IlcAnyArray array)`

This constructor creates an array of `size` constrained set variables. The index range of the array is `[0 size)`, where 0 is included but `size` is excluded. The argument `size` must be strictly greater than 0 (zero). Each constrained variable has no required elements; its possible elements are the pointers in `array`, an array of pointers.

`public **IlcAnySetVarArray**(IloSolver solver, IlcInt size ILCPARAM, const IlcAnySetVar v1, const IlcAnySetVar v2)`

This constructor creates an array of `size` elements. The argument `size` must be strictly greater than 0 (zero). The elements must be successive. If `size` is different from the number of instances of `IlcAnySetVar` passed to the constructor, then the behavior of this constructor is undefined and unlikely to be what you want.

Methods

`public IlcAnySetVarArray **getCopy**(IloSolver solver) const`

This member function returns a copy of the invoking array of constrained set variables and associates that copy with the solver indicated by `solver`.

`public IlcIntSetVarArrayI * **getImpl**() const`

This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

`public const char * **getName**() const`

This member function returns the name of the invoking object.

`public IloSolver **getSolver**() const`

This member function returns an instance of `IloSolver` associated with the invoking object.

`public IloSolverI * **getSolverI**() const`

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

`public void **operator=**(const IlcAnySetVarArray & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

```
public IlcAnySetVar & operator[](IlcInt index) const
```

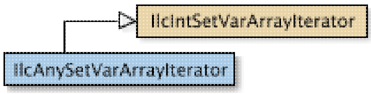
This operator returns a reference to the element at rank `i`. This operator can be used for accessing (that is, simply reading) the element or for modifying (that is, writing) it.

```
public void setName(const char * name) const
```

This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

Class IlcAnySetVarArrayIterator

Definition file: ilsolver/ilcset.h
Include file: <ilsolver/ilosolver.h>



Instances of the class `IlcAnySetVarArrayIterator` traverse the values of an array of *sets* of constrained enumerated variables.

For more information, see the concept Iterator.

See Also: IlcAnySetVar, IlcAnySetVarArray, IlcIntSetVarArrayIterator

Constructor and Destructor Summary	
`public`	`IlcAnySetVarArrayIterator(const IlcAnySetVarArray array)`

Method Summary	
`public IlcBool`	`next(IlcAnySetVar & variable)`

Inherited Methods from `IlcIntSetVarArrayIterator`
`next`

Constructors and Destructors

`public` **`IlcAnySetVarArrayIterator`**`(const IlcAnySetVarArray array)`

This constructor creates an iterator to traverse the values belonging to an array of sets of constrained enumerated variables. This iterator lets you iterate forward over the complete index range of the array.

Methods

`public IlcBool` **`next`**`(IlcAnySetVar & variable)`

This member function takes a reference to a constrained enumerated *set* variable and returns a Boolean value. It begins with the first element. It returns `IlcFalse` if there is no other element on which to iterate and `IlcTrue` otherwise. When it returns `IlcTrue`, it writes the next element of the iterator (forward iteration) to the argument.

Class IlcAnyToIntExpFunction

Definition file: ilsolver/accessor.h
Include file: <ilsolver/ilosolver.h>

It is sometimes useful to associate a constrained variable with an element of the domain of a constrained variable. If the elements of a domain are objects, the associated values can correspond to a specific constrained attribute of these objects.

The following constraints and expressions use this kind of indirection: `IlcSum`, `IlcMin`, `IlcMax`, and `IlcUnion`.

`IlcAnyToIntExpFunction` is the handle class of the object that makes the correspondence between an object element and a constrained integer expression or variable.

An object of this handle class uses the virtual member function IlcIntToIntExpFunctionI::getValue from its implementation class to obtain the associated constrained variable or expression of an element of a domain.

See Also: IlcIntToIntExpFunctionI

Constructor Summary	
public	IlcAnyToIntExpFunction()
public	IlcAnyToIntExpFunction(IlcIntToIntExpFunctionI * impl)

Method Summary	
public IlcIntToIntExpFunctionI *	getImpl() const
public IlcIntExp	getValue(IlcAny elt) const
public void	operator=(const IlcAnyToIntExpFunction & h)

Constructors

public **IlcAnyToIntExpFunction**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IlcAnyToIntExpFunction**(IlcIntToIntExpFunctionI * impl)

This constructor creates a handle object from a pointer to an implementation object.

Methods

public IlcIntToIntExpFunctionI * **getImpl**() const

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

```
public IlcIntExp getValue(IlcAny elt) const
```

This member function returns the integer value associated with the object element `elt`.

```
public void operator=(const IlcAnyToIntExpFunction & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IlcAnyToIntFunction

Definition file: ilsolver/accessor.h
Include file: <ilsolver/ilosolver.h>

It is sometimes useful to associate a constrained variable with an element of the domain of a constrained variable. If the elements of a domain are objects, the associated values can correspond to a specific constrained attribute of these objects.

The following constraints and expressions use this kind of indirection: `IlcSum`, `IlcMin`, `IlcMax`, and `IlcUnion`.

`IlcAnyToIntFunction` is the handle class of the object that makes the correspondence between an object element and an integer value.

An object of this handle class uses the virtual member function IlcIntToIntExpFunctionI::getValue from its implementation class to obtain the associated constrained variable or expression of an element of a domain.

See Also: IlcIntToIntExpFunctionI

Constructor Summary	
public	IlcAnyToIntFunction()
public	IlcAnyToIntFunction(IlcIntToIntFunctionI * impl)

Method Summary	
public IlcIntToIntFunctionI *	getImpl() const
public IlcInt	getValue(IlcAny elt) const
public void	operator=(const IlcAnyToIntFunction & h)

Constructors

`public` **`IlcAnyToIntFunction`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcAnyToIntFunction`**`(IlcIntToIntFunctionI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

Methods

`public IlcIntToIntFunctionI *` **`getImpl`**`() const`

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

`public IlcInt` **`getValue`**`(IlcAny elt) const`

This member function returns the integer value associated with the object element `elt`.

`public void` **`operator=`**`(const IlcAnyToIntFunction & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IlcAnyTupleSet

Definition file: ilsolver/ilcany.h
Include file: <ilsolver/ilosolver.h>

A tuple is an ordered set of values represented by an array. A *set* of tuples is represented by an instance of `IlcAnyTupleSet`. That is, the elements of a tuple *set* are tuples. The number of values in a tuple is known as the *arity* of the tuple, and the arity of the tuples in a set is called the *arity* of the set. (In contrast, the number of tuples in the set is known as the *cardinality* of the set.)

As a handle class, `IlcAnyTupleSet` manages certain set operations efficiently. In particular, elements can be added to such a set. It is also possible to search a given set with the member function `IlcAnyTupleSet::isIn` to see whether or not the set contains a given element.

In addition, a set of tuples can represent a constraint defined on a constrained variable, either as the set of *allowed* combinations of values of the constrained variable on which the constraint is defined, or as the set of *forbidden* combinations of values.

There are a few conventions governing tuple sets:

- When you create the set, you must specify the arity of the tuple-elements it contains.
- You use the member function `IlcAnyTupleSet::add` to add tuples to the set.
- Before searching to determine whether or not a tuple belongs to a given set, you must *close* the set by calling the member function `IlcAnyTupleSet::close`. This operation—closing the set—improves the performance of certain other operations on the set.

Solver will throw an exception (an instance of `IloSolver::SolverErrorException`) if you attempt:

- to add a tuple with a different number of variables from the arity of the set;
- to add an element to a set that has already been closed;
- to search for a tuple with an arity different from the set arity;
- to search for a tuple in a set that has not been closed yet;
- to define a constraint on a tuple set that has not been closed already.

You do not have to worry about memory allocation. If you respect these conventions, Solver manages allocation and de-allocation transparently for you.

See Also: IlcAnyArray, IlcTableConstraint

Constructor Summary	
`public`	`IlcAnyTupleSet()`
`public`	`IlcAnyTupleSet(IlcECSetOfSharedTupleI * impl)`
`public`	`IlcAnyTupleSet(IloSolver solver, IlcInt arity)`

Method Summary	
`public void`	`add(IlcAnyArray tuple) const`
`public void`	`close() const`
`public IlcECSetOfSharedTupleI *`	`getImpl() const`
`public IlcBool`	`isClosed() const`
`public IlcBool`	`isIn(IlcAnyArray tuple) const`
`public void`	`operator=(const IlcAnyTupleSet & h)`

public void	setBigTuple() const
public void	setHoloTuple() const
public void	setSimpleTuple() const

Constructors

public **IlcAnyTupleSet**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IlcAnyTupleSet**(IlcECSetOfSharedTupleI * impl)

This constructor creates a handle object from a pointer to an implementation object.

public **IlcAnyTupleSet**(IloSolver solver, IlcInt arity)

This constructor creates a set of tuples (an instance of the class `IlcAnyTupleSet`) with the arity indicated by `arity`.

Methods

public void **add**(IlcAnyArray tuple) const

This member function adds a tuple represented by the array `tuple` to the invoking set. If you attempt to add an element that is already in the set, that element will *not* be added again. Added elements are not copied; that is, there is no memory duplication. Solver will throw an exception (an instance of `IloSolver::SolverErrorException`) if the set has already been closed. Solver also will throw an exception if the size of the array is not equal to the arity of the invoking set.

public void **close**() const

This member function closes the invoking set. That is, it states that all the tuples in the set are known so efficient data structures can be defined and exploited.

public IlcECSetOfSharedTupleI * **getImpl**() const

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

public IlcBool **isClosed**() const

This member function returns `IlcTrue` if the invoking set has been closed. Otherwise, it returns `IlcFalse`.

public IlcBool **isIn**(IlcAnyArray tuple) const

This member function returns `IlcTrue` if tuple belongs to the invoking set. Otherwise, it returns `IlcFalse`. Solver will throw an exception (an instance of `IloSolver::SolverErrorException`) if the set has not yet been closed. Solver will also throw an exception if the size of the array is not equal to the arity of the invoking set.

`public void` **`operator=`**`(const IlcAnyTupleSet & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

`public void` **`setBigTuple`**`() const`

This member function states that the tuples in the set will be compile in a specific data structure. It must be called before close().

`public void` **`setHoloTuple`**`() const`

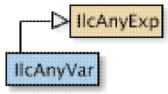
This member function states that the tuples in the set will be compile in a specific data structure. It must be called before close().

`public void` **`setSimpleTuple`**`() const`

This member function states that the tuples in the set will be compile in a specific data structure (the one by default). It must be called before close().

Class IlcAnyVar

Definition file: ilsolver/anyexp.h
Include file: <ilsolver/ilosolver.h>



In order to state constraints on arbitrary objects, Solver defines classes of constrained enumerated variables and expressions. `IlcAnyVar`, the class of constrained enumerated variables, derives from its root class, `IlcAnyExp`, the class of constrained enumerated expressions.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Backtracking and Reversibility

All the member functions and operators defined for this class and capable of modifying constrained variables are *reversible*. In particular, the changes made by constraint-posting functions are made with reversible assignments. Thus, the value, the domain, and the constraints posted on any constrained variable are restored when Solver backtracks.

Domain-Delta and Propagation

When a propagation event is triggered for a constrained variable, the variable is pushed into the propagation queue if it was not already in the queue. Moreover, the modifications of the domain of the constrained variable are stored in a special set called the *domain-delta*. This domain-delta can be accessed during the propagation of the constraints posted on that variable. When all the constraints posted on that variable have been processed, then the domain-delta is cleared. If the variable is modified again, then the whole process begins again. The state of the domain-delta is reversible.

See Also: IlcAnyExp, IlcAnyExpIterator, IlcAnyVarArray, IlcAnyVarDeltaIterator

Constructor Summary	
`public`	`IlcAnyVar(IloSolver solver, const IlcAnyArray values, const char * name=0)`
`public`	`IlcAnyVar()`
`public`	`IlcAnyVar(IlcIntExpI * impl)`
`public`	`IlcAnyVar(const IlcAnyExp exp)`

Method Summary	
`public IlcBool`	`isInDelta(IlcAny value) const`
`public IlcBool`	`isInProcess() const`
`public void`	`operator=(const IlcAnyExp & exp)`
`public void`	`operator=(const IlcAnyVar & exp)`

Inherited Methods from `IlcAnyExp`
`getCopy, getImpl, getName, getObject, getSize, getSolver, getSolverI, getValue, isBound, isInDomain, operator=, removeDomain, removeDomain, removeValue, setDomain, setDomain, setDomain, setName, setObject, setValue, whenDomain, whenValue`

Constructors

`public IlcAnyVar(IloSolver solver, const IlcAnyArray values, const char * name=0)`

This constructor creates a constrained enumerated variable with a domain containing exactly those pointers that belong to `values`, an array of pointers. The optional argument `name`, if provided, becomes the name of the constrained enumerated variable.

`public IlcAnyVar()`

This constructor creates a constrained enumerated variable which is empty, that is, whose handle pointer is null. This object must then be assigned before it can be used, exactly as when you, as a developer, declare a pointer.

`public IlcAnyVar(IlcIntExpI * impl)`

This constructor creates a handle object (an instance of the class `IlcAnyVar`) from a pointer to an object (an instance of the implementation class `IlcIntExpI`).

`public IlcAnyVar(const IlcAnyExp exp)`

To transform a constrained enumerated *expression* (which computes its domain from its subexpressions) into a constrained enumerated *variable* (which stores its domain), you can use this constructor. It associates a domain with the constrained enumerated expression `exp`. This expression thus becomes a constrained enumerated variable. Moreover, the newly created constrained enumerated variable points to the same implementation object as `exp`. (You can also use the assignment operator for the same purpose.)

Methods

`public IlcBool isInDelta(IlcAny value) const`

This member function returns `IlcTrue` if the argument `value` belongs to the domain-delta of the invoking constrained variable. This member function can be applied only to the variable currently in process.

`public IlcBool isInProcess() const`

This member function returns `IlcTrue` if the invoking constrained variable is currently being processed by the constraint propagation algorithm. Only one variable can be in process at a time.

`public void operator=(const IlcAnyExp & exp)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the argument `exp`. After the execution of this operator, the invoking object and the `exp` object both point to the same implementation object. Moreover, this assignment operator associates a domain with the constrained enumerated expression `exp`, which is thus transformed into a constrained enumerated variable.

`public void operator=(const IlcAnyVar & exp)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the argument `exp`. After the execution of this operator, the invoking object and the `exp` object both point to the same implementation object. This assignment operator has no effect on its argument.

Class IlcAnyVarArray

Definition file: ilsolver/anyexp.h
Include file: <ilsolver/ilosolver.h>

The class `IlcAnyVarArray` is the class for an array of instances of `IlcAnyVar`. Three integers—`indexMin`, `indexMax`, and `indexStep`—play an important role in such an array of constrained enumerated variables. The index of those variables ranges from `indexMin`, inclusive, to `indexMax`, exclusive, in steps of `indexStep`. The index of the first variable in the array is `indexMin`; the second one is `indexMin+indexStep`, and so forth. The quantity indicated by `indexMax-indexMin` must be a multiple of `indexStep`.

Generic Constraints

The array makes it easier to implement *generic constraints*. In this context, a generic constraint is a constraint that applies to all of the variables in the array. Member functions of the array class are available to post such generic constraints. A generic constraint is then allocated and recorded only once for all the variables in the array. This fact represents a significant economy in memory, compared to allocating and recording one constraint per variable.

Interval Constraints

Arrays of constrained variables also allow you to define *interval constraints* which propagate in a global way when the domains of one or more constrained variables in the array are modified. Propagation is then performed through a goal. Member functions such as `whenValueInterval` or `whenDomainInterval` associate goals with propagation events for this purpose.

Reversibility

All the functions and member functions capable of modifying arrays of constrained enumerated variables are *reversible*. In particular, when modifiers and functions posting constraints are called, the state before their call will be saved by Solver.

Empty Handle 0r Null Array

It is possible to create a null array, or in other words, an empty handle. When you do so, only these operations are allowed on that null array:

- **copy**: You can assign the null array to a new array.
- **access its size**: The member function `getSize` for the null array returns 0 (zero).
- **create an iterator**: You can create an iterator to traverse the null array. The member function `ok` returns `IlcFalse` for a null array.

Attempts to access a null array in any other way will throw an exception (an instance of `IloSolver::SolverErrorException`).

See Also: IlcAnyVar, IlcAnyVarArrayIterator, IlcIndex, operator<<

Constructor Summary	
public	`IlcAnyVarArray()`
public	`IlcAnyVarArray(IlcIntVarArrayI * impl)`
public	`IlcAnyVarArray(IloSolver solver, IlcInt size)`
public	`IlcAnyVarArray(IloSolver s, IlcInt indexMin, IlcInt indexMax, IlcInt indexStep, const IlcAnyVar prototype)`

Method Summary	
public IlcAnyVarArray	getCopy(IloSolver solver) const
public IlcInt	getDomainIndexMax() const
public IlcInt	getDomainIndexMin() const
public IlcIntVarArrayI *	getImpl() const
public IlcInt	getIndexMax() const
public IlcInt	getIndexMin() const
public IlcInt	getIndexStep() const
public IlcInt	getIndexValue() const
public const char *	getName() const
public IlcInt	getSize() const
public IloSolver	getSolver() const
public IloSolverI *	getSolverI() const
public IlcInt	getValueIndexMax() const
public IlcInt	getValueIndexMin() const
public IlcAnyVar	getVariable(IlcInt index, IlcBool before=IlcFalse) const
public void	operator=(const IlcAnyVarArray & h)
public IlcAnyExp	operator[](IlcIntExp var) const
public IlcAnyExp	operator[](IlcIndex & I) const
public IlcAnyVar &	operator[](IlcInt index) const
public void	setName(const char * name) const
public void	whenDomain(const IlcDemon demon)
public void	whenDomainInterval(const IlcDemon demon)
public void	whenValue(const IlcDemon demon)
public void	whenValueInterval(const IlcDemon demon)

Constructors

public **IlcAnyVarArray**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IlcAnyVarArray**(IlcIntVarArrayI * impl)

This constructor creates a handle object from a pointer to an implementation object.

public **IlcAnyVarArray**(IloSolver solver, IlcInt size)

This constructor creates an uninitialized array of length `size`. The argument `size` must be strictly greater than 0 (zero). The index range of the array is `[0 size)`. Each element of the array must be assigned before the array can be used.

public **IlcAnyVarArray**(IloSolver s, IlcInt indexMin, IlcInt indexMax, IlcInt indexStep, const IlcAnyVar prototype)

This constructor creates an array of copies of the given constrained enumerated variable `prototype`. Each copy initially has the same domain as `prototype` currently has, but the copies do not share the constraints of `prototype`. That is, the copies are independent. The index range of the array is `[indexMin indexMax)`. The step of the array is `indexStep`. Solver will throw an exception (an instance of `IloSolver::SolverErrorException`) if any of the following conditions occur:

- `indexMin` is not strictly less than `indexMax`;
- `indexStep` is not strictly positive;
- `(indexMax-indexMin)` is not a multiple of `indexStep`.

This constructor keeps no pointer to the `prototype` variable. That variable may be an automatic object allocated on the C++ stack.

Methods

`public IlcAnyVarArray` **`getCopy`**`(IloSolver solver) const`

This member function returns a copy of the invoking array of constrained variables and associates that copy with the solver indicated by `solver`.

`public IlcInt` **`getDomainIndexMax`**`() const`

When it is called during the execution of a constraint or a demon associated with an array by the member function `whenDomainInterval`, this member function returns the maximum of the range of the array `[indexMin indexMax)` over which some removal of values has occurred. The returned value of this member function is not meaningful outside the execution of a goal associated with the array by the member function `whenDomainInterval`.

`public IlcInt` **`getDomainIndexMin`**`() const`

When it is called during the execution of a constraint or a demon associated with an array by the member function `whenDomainInterval`, this member function returns the minimum of the range of the array `[indexMin indexMax)` over which some removal of values has occurred. The returned value of this member function is not meaningful outside the execution of a goal associated with the array by the member function `whenDomainInterval`.

`public IlcIntVarArrayI *` **`getImpl`**`() const`

This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

`public IlcInt` **`getIndexMax`**`() const`

This member function returns the maximal index of the invoking array of constrained enumerated variables.

`public IlcInt` **`getIndexMin`**`() const`

This member function returns the minimal index of the invoking array of constrained enumerated variables.

`public IlcInt` **`getIndexStep`**`() const`

This member function returns the index step of the invoking array of constrained enumerated variables. The meaning of this index step is that the indexed variable value may change only at indices equal to `(getIndexMin() + i * getIndexStep())`.

`public IlcInt` **`getIndexValue`**`() const`

When it is called during the execution of a constraint or a demon associated with an array by the member functions `whenValue`, `whenDomain`, or `whenRange`, this member function returns the index in the invoking array of the constrained variable that triggered the propagation event. Calling this member function outside the execution of the goal will throw an exception (an instance of `IloSolver::SolverErrorException`) with the message “`unbound index`”.

`public const char *` **`getName`**`() const`

This member function returns the name of the invoking object.

`public IlcInt` **`getSize`**`() const`

This member function returns the number of variables in the invoking array.

`public IloSolver` **`getSolver`**`() const`

This member function returns an instance of `IloSolver` associated with the invoking object.

`public IloSolverI *` **`getSolverI`**`() const`

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

`public IlcInt` **`getValueIndexMax`**`() const`

When it is called during the execution of a constraint or a demon associated with an array by the member function `whenValueInterval`, this member function returns the maximum of the range of the array `[indexMin indexMax)` over which some binding has occurred. The returned value of this member function is not meaningful outside the execution of a goal associated with the array by the member function `whenValueInterval`.

`public IlcInt` **`getValueIndexMin`**`() const`

When it is called during the execution of a constraint or a demon associated with an array by the member function `whenValueInterval`, this member function returns the minimum of the range of the array `[indexMin indexMax)` over which some binding has occurred. The returned value of this member function is not meaningful outside the execution of a goal associated with the array by the member function `whenValueInterval`.

`public IlcAnyVar` **`getVariable`**`(IlcInt index, IlcBool before=IlcFalse) const`

This member function returns the variable corresponding to the given `index` in the invoking array of constrained enumerated variables. However, if `before` is `IlcTrue`, then `getVariable` returns the variable *before* the variable at the given `index`. Solver will throw an exception (an instance of `IloSolver::SolverErrorException`) with the message "`bad index`" if the given `index` is not a valid one for the invoking array of constrained enumerated variables. Solver will throw the same exception if `index=0` and `before=IlcTrue`.

If `t` is an array of constrained enumerated variables, then these three expressions return the same value:

```
t.getVariable(index, IlcFalse)

t.getVariable(index + 1, IlcTrue)

t[index]
```

`public void` **`operator=`**`(const IlcAnyVarArray & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

`public IlcAnyExp` **`operator[]`**`(IlcIntExp var) const`

This subscripting operator returns a constrained enumerated expression. For clarity, let's call `A` the invoking array. When `var` is bound to the value `i`, then the domain of the variable is the domain of `A[i]`. More generally, the domain of the variable is the union of the domains of the expressions `A[i]` where the `i` are in the domain of `var`.

`public IlcAnyExp` **`operator[]`**`(IlcIndex & I) const`

This operator returns a generic variable, which is a constrained variable representing an element of the array. That generic variable is said to "stem from" the index `i`.

`public IlcAnyVar &` **`operator[]`**`(IlcInt index) const`

This subscripting operator returns a reference to a constrained enumerated variable corresponding to the given `index` in the invoking array of constrained enumerated variables. Solver will throw an exception (an instance of `IloSolver::SolverErrorException`) with the message "`bad index`" if the given `index` is not a valid one for the invoking array of constrained integer variables.

`public void` **`setName`**`(const char * name) const`

This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

`public void` **`whenDomain`**`(const IlcDemon demon)`

This member function associates `demon` with the domain propagation event of every variable in the invoking array. Whenever a value is removed from the domain of any of the variables in the invoking array, the demon will be executed immediately.

When the demon is executed, the index of the constrained variable that triggered the domain event can be known by a call to the member function `getIndexValue`.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. When a value is removed from the domain of any of the variables in the invoking array, the constraint will be propagated.

```
public void whenDomainInterval(const IlcDemon demon)
```

This member function associates `demon` with the domain propagation event of every variable in the invoking array. It specifies that a given demon reacts globally to modifications of the domains of a collection of variables in the array. Whenever a domain propagation event or a series of such events occurs, the demon will be executed immediately.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. Whenever a domain propagation event or a series of such events occurs, the constraint will be propagated.

A call to the demon signifies that *some* removal of values from domain(s) occurred over the index range `[indexMin indexMax)`. It does *not* mean that values have been removed from the domains of all the variables in the range.

```
public void whenValue(const IlcDemon demon)
```

This member function associates `demon` with the value propagation event of every variable in the invoking array. When one of the variables in the array receives a value, the demon will be executed immediately.

When the demon is executed, the index of the bound constrained variable can be known by a call to the member function `getIndexValue`.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. When one of the variables in the array receives a value, the constraint will be propagated.

```
public void whenValueInterval(const IlcDemon demon)
```

This member function associates `demon` with the value propagation event of every variable in the invoking array. It specifies that a given demon reacts globally to the binding of a collection of variables in the array. When a value propagation event or a series of such events occurs, the demon will be executed immediately.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. When a value propagation event or a series of such events occurs, the constraint will be propagated.

A call to the demon signifies that *some* variable binding occurred over the index range `[indexMin indexMax)`. It does *not* mean that all the variables in the range have been bound.

Class IlcAnyVarArrayIterator

Definition file: ilsolver/anyexp.h
Include file: <ilsolver/ilosolver.h>

Solver provides iterators to traverse the values of an array of constrained enumerated variables. Those iterators are instances of the class `IlcAnyVarArrayIterator`. An instance of that class iterates forward over a subinterval `[indexMin indexMax)` of the index range of the array.

For more information, see the concept Iterator.

See Also: IlcAnyVar, IlcAnyVarArray

Constructor and Destructor Summary	
public	IlcAnyVarArrayIterator(const IlcAnyVarArray array)

Method Summary	
public IlcBool	next(IlcAnyVar & variable)
public IlcBool	ok() const

Constructors and Destructors

`public` **`IlcAnyVarArrayIterator`**`(const IlcAnyVarArray array)`

This constructor creates an iterator to traverse the values belonging to the array of constrained enumerated variables. This iterator lets you iterate forward over the complete index range of the array.

Methods

`public IlcBool` **`next`**`(IlcAnyVar & variable)`

This member function takes a reference to a constrained enumerated variable and returns a Boolean value. It begins with the first element. It returns `IlcFalse` if there is no other element on which to iterate and `IlcTrue` otherwise. When it returns `IlcTrue`, it writes the next element of the iterator (forward iteration) to the argument.

`public IlcBool` **`ok`**`() const`

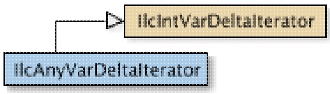
This member function returns `IlcTrue` if there is a current element and the invoking iterator points to it. Otherwise, it returns `IlcFalse`.

To traverse the elements of a finite set of pointers, use the following code:

```
IlcAny val;
for(IlcAnySetIterator iter(set); iter.ok(); ++iter){
              val = *iter;
              // do something with val
}
```

Class IlcAnyVarDeltaIterator

Definition file: ilsolver/anyexp.h
Include file: <ilsolver/ilosolver.h>



An instance of the class `IlcAnyVarDeltaIterator` is an iterator that traverses the values belonging to the domain-delta of a constrained enumerated variable (that is, an instance of `IlcAnyVar`).

For more information, see the concepts Propagation, Iterator, and Domain-Delta.

See Also: IlcAnyVar

Constructor and Destructor Summary	
public	IlcAnyVarDeltaIterator(const IlcAnyVar var)

Method Summary	
public IlcBool	ok() const
public IlcAny	operator*() const
public IlcAnyVarDeltaIterator &	operator++()

Inherited Methods from `IlcIntVarDeltaIterator`
ok, operator*, operator++

Constructors and Destructors

public **IlcAnyVarDeltaIterator**(const IlcAnyVar var)

This constructor creates an iterator associated with `var` to traverse the values belonging to the domain-delta of `var`.

Methods

public IlcBool **ok**() const

This member function returns `IlcTrue` if there is a current element and the invoking iterator points to it. Otherwise, it returns `IlcFalse`.

To traverse the values belonging to the domain-delta of a constrained enumerated variable, use the following code:

```
IlcAny val;
for(IlcAnyVarDeltaIterator iter(set); iter.ok(); ++iter){
              val = *iter;
              // do something with val
}
```

public IlcAny **operator***() const

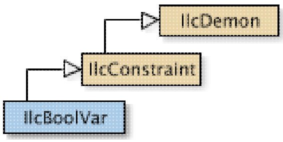
This operator returns the current element, the one to which the invoking iterator points.

```
public IlcAnyVarDeltaIterator & operator++()
```

This operator advances the iterator to point to the next value in the domain-delta of the constrained enumerated variable.

Class IlcBoolVar

Definition file: ilsolver/numi.h
Include file: <ilsolver/ilosolver.h>



Constrained Boolean variables are variables whose possible values are `IlcTrue` and `IlcFalse`. These variables are instances of the class `IlcBoolVar`, which is a subclass of the class `IlcConstraint`. This class inherits the member functions, `isTrue` and `isFalse`, as well as the operators defined for `IlcConstraint`, of course.

A constrained Boolean *expression* can be transformed into a constrained Boolean *variable* by means of the class constructor or the assignment operator. The effect of either of those is to associate a *domain* with the constrained Boolean expression.

See Also: IlcBoolAbstraction, IlcBoolVarArray, IlcConstraint, IlcGoal

Constructor Summary	
public	`IlcBoolVar(IloSolver solver, const char * name=0)`
public	`IlcBoolVar(const IlcConstraint exp)`
public	`IlcBoolVar()`

Method Summary	
public IlcBool	`isBound()`
public void	`operator=(const IlcBoolVar & exp)`
public void	`operator=(const IlcConstraint & exp)`

Inherited Methods from `IlcConstraint`
`getImpl, getName, getObject, getParentDemon, getSolver, isFalse, isTrue, setName, setObject`

Inherited Methods from `IlcDemon`
`getConstraint, getImpl, getSolver, operator=`

Constructors

`public` **`IlcBoolVar`**`(IloSolver solver, const char * name=0)`

This constructor creates a constrained Boolean variable with a domain containing `IlcTrue` and `IlcFalse`. The optional argument `name`, if provided, becomes the name of the constrained Boolean variable.

`public` **`IlcBoolVar`**`(const IlcConstraint exp)`

This constructor associates a domain with the constrained Boolean expression, `exp`. That expression thus becomes a constrained Boolean variable. Moreover, the newly created constrained Boolean variable points to the same implementation object as `exp`.

```
public IlcBoolVar()
```

This constructor creates a constrained Boolean variable which is empty, that is, whose handle pointer is null. This object must then be assigned before it can be used, exactly as when you declare a pointer.

Methods

```
public IlcBool isBound()
```

This member function indicates whether or not its invoking object is bound to a value in its domain.

```
public void operator=(const IlcBoolVar & exp)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the argument `exp`. After the execution of this operator, the invoking object and `exp` both point to the same implementation object. This assignment operator has no effect on its argument.

```
public void operator=(const IlcConstraint & exp)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the argument `exp`. After the execution of this operator, the invoking object and the `exp` object both point to the same implementation object. Moreover, this assignment operator associates a domain with the constrained Boolean expression `exp`, which is thus transformed into a constrained Boolean variable.

Class IlcBoolVarArray

Definition file: ilsolver/numi.h
Include file: <ilsolver/ilosolver.h>

The class `IlcBoolVarArray` is the class for an *array* of instances of `IlcBoolVar`.

Generic Constraints

The array makes it easier to implement *generic* constraints. In this context, a generic constraint is a constraint that applies to all of the variables in the array. Member functions of the array class are available to post such generic constraints. A generic constraint is then allocated and recorded only once for all the variables in the array. This fact represents a significant economy in memory, compared to allocating and recording one constraint per variable.

Backtracking and Reversibility

All the functions and member functions capable of modifying arrays of constrained Boolean variables are *reversible*. In particular, when modifiers and functions that post constraints are called, the state before their call will be saved by Solver.

Empty Handle or Null Array

It is possible to create a null array, or in other words, an empty handle. When you do so, only these operations are allowed on that null array:

- **copy**: You can assign the null array to a new array.
- **access its size**: The member function `getSize` for the null array returns 0 (zero).
- **create an iterator**: You can create an iterator to traverse the null array. The member function `ok` returns `IlcFalse` for a null array.

Attempts to access a null array in any other way will throw an exception (an instance of `IloSolver::SolverErrorException`).

See Also: IlcAbstraction, IlcBoolAbstraction, IlcIndex

Constructor Summary	
`public`	`IlcBoolVarArray()`
`public`	`IlcBoolVarArray(IlcBoolVarArrayI * impl)`
`public`	`IlcBoolVarArray(const IlcBoolVarArray & array)`
`public`	`IlcBoolVarArray(IloSolver solver, IlcInt size)`

Method Summary	
`public void`	`display(ostream & stream) const`
`public IlcBoolVar &`	`getElem(IlcInt index) const`
`public IlcBoolVarArrayI *`	`getImpl() const`
`public IlcInt`	`getSize() const`
`public IloSolver`	`getSolver() const`
`public void`	`operator=(const IlcBoolVarArray & array)`
`public IlcBoolVar &`	`operator[](IlcInt index) const`

Constructors

`public IlcBoolVarArray()`

This constructor creates an uninitialized array of constrained Boolean variables. The index range of the array is undefined. The array must be assigned before it can be used.

`public IlcBoolVarArray(IlcBoolVarArrayI * impl)`

This constructor creates a handle object (an instance of the class `IlcBoolVarArray`) from a pointer to an object (an instance of the implementation class `IlcBoolVarArrayI`).

`public IlcBoolVarArray(const IlcBoolVarArray & array)`

This copy constructor creates a new instance of `IlcBoolVarArray`. After execution of this constructor, both the newly created array and `array` point to the same implementation object.

`public IlcBoolVarArray(IloSolver solver, IlcInt size)`

This constructor creates an instance of `IlcBoolVarArray` and adds it to those belonging to `solver`. The parameter `size`, of course, indicates the size of the new array.

Methods

`public void display(ostream & stream) const`

This member function is called by the operator `<<`. The name of the invoking array, if it has been assigned with `setName`, or the string `IlcBoolVarArrayI`, otherwise, will be printed on the given output stream, followed by the display of all the elements of the array enclosed by brackets.

`public IlcBoolVar & getElem(IlcInt index) const`

This member function returns a reference to the constrained Boolean variable located at `index` in the invoking array.

`public IlcBoolVarArrayI * getImpl() const`

This member function returns a pointer to the implementation object of the invoking array.

`public IlcInt getSize() const`

This member function returns an integer that indicates the size of the invoking array.

`public IloSolver getSolver() const`

This member function returns an instance of `IloSolver` associated with the invoking array.

`public void` **`operator=`**`(const IlcBoolVarArray & array)`

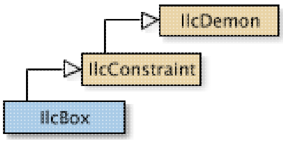
This assignment operator copies `array` into the invoking constrained Boolean array by assigning an address to the handle pointer of the invoking object. That address is the location of the implementation object of the argument `array`.

`public IlcBoolVar &` **`operator[]`**`(IlcInt index) const`

This subscripting operator returns a reference to a constrained Boolean variable corresponding to the given `index` in the invoking array of constrained Boolean variables. Solver will throw an exception (an instance of `IloSolver::SolverErrorException`) with the message "`bad index`" if the given `index` is not a valid one for the invoking array of constrained Boolean variables.

Class IlcBox

Definition file: ilsolver/ilcbox.h
Include file: <ilsolver/ilosolver.h>



Instances of the class `IlcBox` are multidimensional boxes that appear in multidimensional placement problems. To solve packing or placement problems, you may need to be able to place boxes within a given container. In such a situation, both the boxes to place and the container to hold them are instances of the class `IlcBox`.

To specify the containment constraint that a given container holds a given box, use the member function `IlcBox::contains`.

Backtracking and Reversibility

All the member functions and operators defined for this class and capable of modifying constrained variables are reversible. In particular, the changes made by constraint-posting functions are made with reversible assignments. Thus the value of the domain, and the constraints posted on any constrained variable are restored when Solver backtracks.

For more information, see `IloBox`.

See Also: IlcBoxIterator, IlcFilterLevelConstraint, IloBox

Constructor and Destructor Summary	
public	`IlcBox(IlcInt dimensions, IlcIntVarArray origin, IlcIntArray size)`

Method Summary	
public IlcConstraint	`contains(IlcBox box)`
public IlcBool	`doesNotOverlapInDimension(IlcBox box1, IlcBox box2, IlcInt dimension)`
public IlcBool	`doesOverlapInDimension(IlcBox box1, IlcBox box2, IlcInt dimension)`
public IlcBool	`doesPrecedeInDimension(IlcBox box1, IlcBox box2, IlcInt dimension)`
public IlcInt	`getDimensions()`
public IlcBoxI *	`getImpl() const`
public IlcConstraint	`getNotOverlapConstraint()`
public IlcAny	`getObject() const`
public IlcIntVar	`getOrigin(IlcInt dimension)`
public IlcInt	`getSize(IlcInt dimension)`
public IloSolver	`getSolver() const`
public IlcBool	`isContained(IlcBox box)`
public IlcConstraint	`notOverlapInDimension(IlcBox box1, IlcBox box2, IlcInt dimension)`
public IlcBool	`notOverlapInDimensionKnown(IlcBox box1, IlcBox box2, IlcInt`

	dimension)
public IlcConstraint	overlapInDimension(IlcBox box1, IlcBox box2, IlcInt dimension)
public IlcBool	overlapInDimensionKnown(IlcBox box1, IlcBox box2, IlcInt dimension)
public IlcConstraint	precedenceInDimension(IlcBox box1, IlcBox box2, IlcInt dimension)
public IlcBool	precedenceInDimensionKnown(IlcBox box1, IlcBox box2, IlcInt dimension)
public void	setObject(IlcAny object)

Inherited Methods from `IlcConstraint`
getImpl, getName, getObject, getParentDemon, getSolver, isFalse, isTrue, setName, setObject

Inherited Methods from `IlcDemon`
getConstraint, getImpl, getSolver, operator=

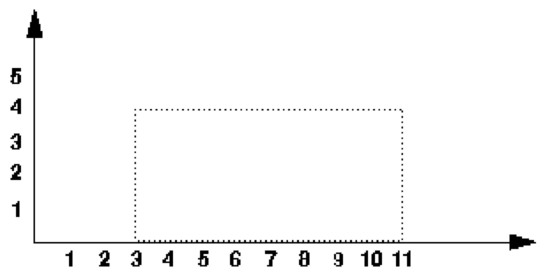
Constructors and Destructors

`public` **`IlcBox`**`(IlcInt dimensions, IlcIntVarArray origin, IlcIntArray size)`

This constructor creates a box according to the specifications passed in the parameters. The parameter `dimensions` indicates the number of dimensions the box has. The arrays `origin` and `size` must contain the same number of elements as the number of dimensions of the box. In dimension *i*, the box extends from `origin[`*i*`]` to `origin[`*i*`] + size[`*i*`]`. For example, the statement

```
IlcBox(2,IlcIntVarArray(s,2,IlcIntVar(s,3,3),IlcIntVar(s,0,0)), IlcIntArray(s,2,8,4));
```

creates a box as shown in the illustration below.



Methods

`public IlcConstraint` **`contains`**`(IlcBox box)`

This member function creates a constraint that requires the invoking box to contain `box`. The parameter `box` should have the same number of dimensions as the invoking box.

This member function is reversible. That is, if a failure causes Solver to backtrack to an earlier choice point, the effect of the `contains` statement is undone automatically.

`public IlcBool` **`doesNotOverlapInDimension`**`(IlcBox box1, IlcBox box2, IlcInt dimension)`

This member function returns `IlcTrue` if the corresponding `notOverlapInDimension` constraint is verified. Otherwise, it returns `IlcFalse`.

`public IlcBool` **`doesOverlapInDimension`**`(IlcBox box1, IlcBox box2, IlcInt dimension)`

This member function returns `IlcTrue` if the corresponding `overlapInDimension` constraint is verified. Otherwise, it returns `IlcFalse`.

`public IlcBool` **`doesPrecedeInDimension`**`(IlcBox box1, IlcBox box2, IlcInt dimension)`

This member function returns `IlcTrue` if the corresponding `precedenceInDimension` constraint is verified. Otherwise, it returns `IlcFalse`.

`public IlcInt` **`getDimensions`**`()`

This member function returns the number of dimensions of the invoking box.

`public IlcBoxI *` **`getImpl`**`() const`

This member function returns the implementation object of the invoking object. You can use this member function to check whether a constraint is empty.

`public IlcConstraint` **`getNotOverlapConstraint`**`()`

This member function returns a constraint that specifies that none of the boxes contained in the invoking box can overlap simultaneously along all of their dimensions.

The filter level of this constraint can be set using `IloSolver::setFilterLevel`. Currently two filter levels are allowed, `IlcLow`, the default, and `IlcBasic`, which is slower but propagates more. You can specify the default filter level of this constraint using the method `IloSolver::setDefaultFilterLevel`.

`public IlcAny` **`getObject`**`() const`

This member function returns the object associated with the invoking box.

`public IlcIntVar` **`getOrigin`**`(IlcInt dimension)`

This member function returns the origin of the invoking box along dimension `dimension`.

`public IlcInt` **`getSize`**`(IlcInt dimension)`

This member function returns the length of the invoking box along dimension `dimension`.

`public IloSolver **getSolver**() const`

This member function returns the solver associated with the invoking box.

`public IlcBool **isContained**(IlcBox box)`

This member function returns `IlcTrue` if the invoking box object is contained in `box`. Otherwise, it returns `IlcFalse`.

`public IlcConstraint **notOverlapInDimension**(IlcBox box1, IlcBox box2, IlcInt dimension)`

This member function specifies the constraint that if `box1` and `box2` are contained in the invoking box, then `box1` and `box2` must not overlap along dimension `dimension`.

`public IlcBool **notOverlapInDimensionKnown**(IlcBox box1, IlcBox box2, IlcInt dimension)`

This member function returns `IlcTrue` if the truth value of the corresponding `notOverlapInDimension` constraint is known. Otherwise, it returns `IlcFalse`.

`public IlcConstraint **overlapInDimension**(IlcBox box1, IlcBox box2, IlcInt dimension)`

This member function specifies the constraint that if `box1` and `box2` are contained in the invoking box, then `box1` and `box2` must overlap along dimension `dimension`.

`public IlcBool **overlapInDimensionKnown**(IlcBox box1, IlcBox box2, IlcInt dimension)`

This member function returns `IlcTrue` if the truth value of the corresponding `overlapInDimension` constraint is known. Otherwise, it returns `IlcFalse`.

`public IlcConstraint **precedenceInDimension**(IlcBox box1, IlcBox box2, IlcInt dimension)`

This member function specifies the constraint that if `box1` and `box2` are contained in the invoking box, then `box1` must precede `box2` in `dimension`.

`public IlcBool **precedenceInDimensionKnown**(IlcBox box1, IlcBox box2, IlcInt dimension)`

This member function returns `IlcTrue` if the truth value of the corresponding `precedenceInDimension` constraint is known. Otherwise, it returns `IlcFalse`.

`public void **setObject**(IlcAny object)`

This member function sets the object associated with the invoking box to `object`.

Class IlcBoxIterator

Definition file: ilsolver/ilcbox.h
Include file: <ilsolver/ilosolver.h>

An instance of this class is an iterator capable of traversing the boxes contained in an instance of `IlcBox`.

See Also: IlcBox

Constructor Summary	
`public`	`IlcBoxIterator(IlcBox box)`

Method Summary	
`public IlcBool`	`next(IlcBox & next_box)`

Constructors

`public` **`IlcBoxIterator`**`(IlcBox box)`

This constructor creates an iterator to traverse the boxes contained in the box `box`.

Methods

`public IlcBool` **`next`**`(IlcBox & next_box)`

This member function advances the iterator to the next box contained in `box`. The `IlcBoxIterator::next` method returns `IlcTrue` if there is a next box and `IlcFalse` otherwise.

Class IlcConstAnyArray

Definition file: ilsolver/anyexp.h
Include file: <ilsolver/ilosolver.h>

`IlcConstAnyArray` is the unchanging, constant array class for the basic enumerated class. You cannot modify the values of elements of such an array. When you build an instance of this class, its constructor systematically copies the array passed to it. This is a handle class. The implementation class for `IlcConstAnyArray` is the undocumented class `IlcConstIntArrayI`. Instances of this class are useful, for example, in the function `IlcTableConstraint` when you want to *share* an array rather than copy it.

See Also: IlcAnyArray, IlcTableConstraint, operator<<

Constructor Summary	
public	IlcConstAnyArray()
public	IlcConstAnyArray(IlcConstIntArrayI * impl)
public	IlcConstAnyArray(IloSolver s, IlcInt size, IlcAny * values)
public	IlcConstAnyArray(IloSolver solver, IlcInt size, const IlcAny exp0, const IlcAny exp...)
public	IlcConstAnyArray(IloSolver solver, IlcAnyArray array)

Method Summary	
public void	display(ostream & str) const
public IlcConstIntArrayI *	getImpl() const
public IlcInt	getSize() const
public IloSolver	getSolver() const
public void	operator=(const IlcConstAnyArray & h)
public IlcAny	operator[](IlcInt i) const

Constructors

public **IlcConstAnyArray**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IlcConstAnyArray**(IlcConstIntArrayI * impl)

This constructor creates a handle object from a pointer to an implementation object.

public **IlcConstAnyArray**(IloSolver s, IlcInt size, IlcAny * values)

This constructor creates a constant array containing the values in the array `values`. The argument `size` must be the length of the array `values`; it must also be strictly greater than 0 (zero). Solver does not keep a pointer to the array `values`.

```
public IlcConstAnyArray(IloSolver solver, IlcInt size, const IlcAny exp0, const
IlcAny exp...)
```

This constructor accepts a variable number of arguments. Its first argument, `size`, indicates the length of the array that this constructor will create; `size` must be the same as the number of instances of `IlcAny` passed as arguments; it must also be strictly greater than 0 (zero). The constructor creates a constant array of the values indicated by the other arguments.

```
public IlcConstAnyArray(IloSolver solver, IlcAnyArray array)
```

This constructor creates a constant version of `array`.

Methods

```
public void display(ostream & str) const
```

This member function is called by the `operator <<`. The string `IlcConstIntArrayI` will be printed on the given output stream, followed by the display of all the elements of the array enclosed by brackets.

```
public IlcConstIntArrayI * getImpl() const
```

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

```
public IlcInt getSize() const
```

This member function returns the number of elements in the array.

```
public IloSolver getSolver() const
```

This member function returns an instance of `IloSolver` associated with the invoking object.

```
public void operator=(const IlcConstAnyArray & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

```
public IlcAny operator[](IlcInt i) const
```

This operator returns the element at index `i`. This operator can be used for accessing (that is, simply reading) the element.

Class IlcConstFloatArray

Definition file: ilsolver/fltexp.h
Include file: <ilsolver/ilosolver.h>

`IlcConstFloatArray` is the unchanging, constant array class for the basic floating-point class. You cannot modify the values of elements of such an array. When you build an instance of this class, its constructor systematically copies the array passed to it. This is a handle class. The implementation class for `IlcConstFloatArray` is the undocumented class `IlcConstFloatArrayI`. Instances of this class are useful, for example, in the function `IlcTableConstraint` when you want to *share* an array rather than copy it.

See Also: IlcFloatArray, IlcTableConstraint, operator<<

Constructor Summary	
`public`	`IlcConstFloatArray()`
`public`	`IlcConstFloatArray(IlcConstFloatArrayI * impl)`
`public`	`IlcConstFloatArray(IloSolver s, IlcInt size, IlcFloat * values)`
`public`	`IlcConstFloatArray(IloSolver s, IlcFloatArray array)`
`public`	`IlcConstFloatArray(IloSolver solver, IlcInt size, IlcFloat exp0, IlcFloat exp...)`
`public`	`IlcConstFloatArray(IloSolver solver, IlcInt size, IlcInt exp0, IlcInt exp...)`

Method Summary	
`public void`	`display(ostream & strm) const`
`public IlcConstFloatArrayI *`	`getImpl() const`
`public IlcInt`	`getSize() const`
`public IloSolver`	`getSolver() const`
`public void`	`operator=(const IlcConstFloatArray & h)`
`public IlcFloat`	`operator[](IlcInt i) const`

Constructors

`public` **`IlcConstFloatArray`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcConstFloatArray`**`(IlcConstFloatArrayI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public` **`IlcConstFloatArray`**`(IloSolver s, IlcInt size, IlcFloat * values)`

This constructor creates a constant array containing the values in the array `values`. The argument `size` must be the length of the array `values`; it must also be strictly greater than 0 (zero). Solver does not keep a pointer to the array `values`.

`public IlcConstFloatArray(IloSolver s, IlcFloatArray array)`

This constructor creates a constant array where the values of the elements cannot be changed from the elements of `array`.

`public IlcConstFloatArray(IloSolver solver, IlcInt size, IlcFloat exp0, IlcFloat exp...)`

This constructor accepts a variable number of arguments. Its first argument, `size`, indicates the length of the array that this constructor will create; `size` must be the same as the number of instances of `IlcFloat` passed as arguments; it must also be strictly greater than 0 (zero). The constructor creates a constant array of the values indicated by the other arguments.

`public IlcConstFloatArray(IloSolver solver, IlcInt size, IlcInt exp0, IlcInt exp...)`

This constructor accepts a variable number of arguments. Its first argument, `size`, indicates the length of the array that this constructor will create; `size` must be the same as the number of instances of `IlcInt` passed as arguments; it must also be strictly greater than 0 (zero). The constructor creates a constant array of the values indicated by the other arguments. The other arguments must all be of the same type. Do not mix the type of elements in a given array.

Methods

`public void display(ostream & strm) const`

This member function is called by the `operator <<`. The string `IlcConstFloatArrayI` will be printed on the given output stream, followed by the display of all the elements of the array enclosed by brackets.

`public IlcConstFloatArrayI * getImpl() const`

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

`public IlcInt getSize() const`

This member function returns the number of elements in the array.

`public IloSolver getSolver() const`

This member function returns an instance of `IloSolver` associated with the invoking object.

`public void operator=(const IlcConstFloatArray & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

```
public IlcFloat operator[](IlcInt i) const
```

This operator returns the element at index `i`. This operator can be used for accessing (that is, simply reading) the element.

Class IlcConstIntArray

Definition file: ilsolver/intexp.h
Include file: <ilsolver/ilosolver.h>

`IlcConstIntArray` is the unchanging, constant array class for the basic integer class. You cannot modify the values of elements of such an array. When you build an instance of this class, its constructor systematically copies the array passed to it. This is a handle class. The implementation class for `IlcConstIntArray` is the undocumented class `IlcConstIntArrayI`. Instances of this class are useful, for example, in the function `IlcTableConstraint` when you want to *share* an array rather than copy it.

See Also: IlcIntArray, IlcTableConstraint, operator<<

Constructor Summary	
public	IlcConstIntArray()
public	IlcConstIntArray(IlcConstIntArrayI * impl)
public	IlcConstIntArray(IloSolver s, IlcIntArray array)
public	IlcConstIntArray(IloSolver s, IlcInt size, IlcInt * values)
public	IlcConstIntArray(IloSolver solver, IlcInt size, IlcInt exp0, IlcInt exp...)

Method Summary	
public void	display(ostream & str) const
public IlcConstIntArrayI *	getImpl() const
public IlcInt	getSize() const
public IloSolver	getSolver() const
public void	operator=(const IlcConstIntArray & h)
public IlcInt	operator[](IlcInt i) const

Constructors

public **IlcConstIntArray**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IlcConstIntArray**(IlcConstIntArrayI * impl)

This constructor creates a handle object from a pointer to an implementation object.

public **IlcConstIntArray**(IloSolver s, IlcIntArray array)

This constructor creates a constant array where the values of the elements cannot be changed from the elements of `array`.

public **IlcConstIntArray**(IloSolver s, IlcInt size, IlcInt * values)

This constructor creates a constant array containing the values in the array `values`. The argument `size` must be the length of the array `values`; it must also be strictly greater than 0 (zero). Solver does not keep a pointer to the array `values`.

```
public IlcConstIntArray(IloSolver solver, IlcInt size, IlcInt exp0, IlcInt exp...)
```

This constructor accepts a variable number of arguments. Its first argument, `size`, indicates the length of the array that this constructor will create; `size` must be the same as the number of instances of `IlcInt` passed as arguments; it must also be strictly greater than 0 (zero). The constructor creates a constant array of the values indicated by the other arguments.

Methods

```
public void display(ostream & str) const
```

This member function is called by the `operator <<`. The string `IlcConstIntArrayI` will be printed on the given output stream, followed by the display of all the elements of the array enclosed by brackets.

```
public IlcConstIntArrayI * getImpl() const
```

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

```
public IlcInt getSize() const
```

This member function returns the number of elements in the array.

```
public IloSolver getSolver() const
```

This member function returns an instance of `IloSolver` associated with the invoking object.

```
public void operator=(const IlcConstIntArray & h)
```

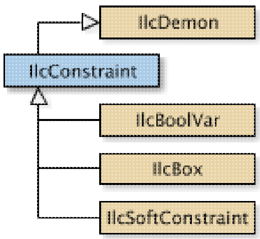
This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

```
public IlcInt operator[](IlcInt i) const
```

This operator returns the element at index `i`. This operator can be used for accessing (that is, simply reading) the element.

Class IlcConstraint

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>



A constraint is an object in Solver. Like other Solver entities, a constraint is implemented by means of two classes: a handle class and an implementation class. In other words, an instance of the class `IlcConstraint` (a handle) contains a data member (the handle pointer) that points to an instance of the class `IlcConstraintI` (its implementation object).

Parent of a Constraint

When you use the member function `IloSolver::add` to add the constraint C to the constraint P, then we say that the constraint P is the *parent* of constraint C.

If there is no other constraint associated with a given constraint as its parent, then we say that the constraint is its own parent, and we also say that it is an *external* constraint. In that case, the member function `IlConstraint::getParentDemon` returns 0 (zero, an empty handle).

Parents of constraints are useful in the trace mechanism of Solver.

Constraints as Boolean Expressions

You might also think of a constraint as a Boolean expression with a value of either `IlcFalse` or `IlcTrue`. The value of the expression depends on the satisfiability of the constraint: if the constraint cannot be violated, then the expression is bound to `IlcTrue`; if the constraint cannot be satisfied, then the expression is bound to `IlcFalse`. We sometimes call a constraint, a *constrained Boolean expression*. These expressions can be constrained themselves, and they can be combined with logical operators: *or*, *and*, and *not*.

Posting Constraints

When you create a constraint, Solver does not automatically take it into account. You must explicitly add the constraint to a model and extract the model for an algorithm.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Constructor Summary	
`public`	`IlcConstraint()`
`public`	`IlcConstraint(const IlcConstraint & c)`
`public`	`IlcConstraint(IlcConstraintI * impl)`

Method Summary	
`public IlcConstraintI *`	`getImpl() const`
`public char *`	`getName() const`
`public IlcAny`	`getObject() const`

public IlcDemon	getParentDemon() const
public IloSolver	getSolver() const
public IlcBool	isFalse() const
public IlcBool	isTrue() const
public void	setName(const char * name) const
public void	setObject(IlcAny object) const

Inherited Methods from `IlcDemon`
getConstraint, getImpl, getSolver, operator=

Constructors

public **IlcConstraint**()

This constructor creates a constraint which is empty, that is, one whose handle pointer is null. This object must then be assigned before it can be used, exactly as when you declare a pointer.

public **IlcConstraint**(const IlcConstraint & c)

This copy constructor creates a reference to a constraint. That constraint and `c` both point to the same implementation object.

public **IlcConstraint**(IlcConstraintI * impl)

This constructor creates a handle object (an instance of the class `IlcConstraint`) from a pointer to an object (an instance of the implementation class `IlcConstraintI`).

Methods

public IlcConstraintI * **getImpl**() const

This member function returns the implementation object of the invoking object. You can use this member function to check whether a constraint is empty.

public char * **getName**() const

This member function returns the name of the invoking handle, if the handle has a name. Otherwise, it returns the empty string.

public IlcAny **getObject**() const

It is possible to associate some object (other than the implementation object) with a handle. This member function accesses such an associated object. It returns a pointer to the associated object, if one exists. Otherwise, the member function returns the null pointer.

`public IlcDemon` **`getParentDemon`**`() const`

When you use the macro `ILCCTDEMON0` to construct a class of demons, then each demon of that class may be associated with a constraint as the *parent* of that constraint. This member function returns the handle of the demon associated with the invoking constraint. If there is no demon explicitly associated as the parent of the invoking constraint, then this member function returns the invoking constraint itself.

`public IloSolver` **`getSolver`**`() const`

This member function returns the solver (a handle) of the invoking constraint.

`public IlcBool` **`isFalse`**`() const`

When this member function returns `IlcTrue`, the invoking constraint cannot be satisfied, no matter what.

For example, consider the constraint `x <= y`. If the domain of `x` is `[0, 10]` and the domain of `y` is `[10, 20]`, then the constraint is necessarily satisfied. In contrast, if the domain of `x` is `[11, 20]` and the domain of `y` is `[0, 10]`, then the constraint is necessarily violated (that is, it cannot be satisfied).

`public IlcBool` **`isTrue`**`() const`

When this member function returns `IlcTrue`, the invoking constraint is necessarily satisfied, no matter what.

`public void` **`setName`**`(const char * name) const`

This member function sets the name of the invoking handle to a copy of the given `name`. This is a reversible action: the new name is allocated on the Solver heap, and the previous name is automatically restored upon backtracking. This member function keeps no pointer to the string `name`.

`public void` **`setObject`**`(IlcAny object) const`

It is possible to associate some object (other than the implementation object) with a handle by means of this member function. If the invoking handle has no associated object, then `object` becomes the associated object. If the invoking handle already has an associated object, Solver will throw an exception (an instance of `IloSolver::SolverErrorException`). The argument `object` must not be the null pointer; otherwise, Solver will throw an exception (an instance of `IloSolver::SolverErrorException`).

Class IlcConstraintAggregator

Definition file: ilsolver/ilosolverhandle.h
Include file: <ilsolver/ilosolver.h>

A constraint aggregator is an object that is attached to an instance of `IloSolver` using the function:

```
void IloSolver::use(IlcConstraintAggregator agg) const;
```

It enhances the instance of IloSolver by improving propagation or by providing additional information.

See Also: IlcAllDiffAggregator, IlcLogicAggregator, IlcLinearCtAggregator, IlcReductionInformation, IlcDegreeInformation, IlcImpactInformation

Class IlcConstraintArray

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>

The class `IlcConstraintArray` is the class for an array of instances of `IlcConstraint`.

It is a handle class. The implementation class for `IlcConstraintArray` is the undocumented class `IlcConstraintArrayI`. An object of this class contains a pointer to a constraint allocated on the Solver heap. Exploiting handles in this way greatly simplifies the programming interface since the handle can then be an automatic object. When you use handles you do not have to worry about memory allocation.

Empty Handle or Null Array

It is possible to create a null array, or in other words, an empty handle. When you do so, only these operations are allowed on that null array:

- **copy**: You can assign the null array to a new array.
- **access its size**: The member function `getSize` for the null array returns 0 (zero).
- **create an iterator**: You can create an iterator to traverse the null array. The member function `ok` returns `IlcFalse` for a null array.

Attempts to access a null array in any other way will throw an exception (an instance of `IloSolver::SolverErrorException`).

See Also: IlcConstraint, operator<<

Constructor Summary	
public	IlcConstraintArray()
public	IlcConstraintArray(IlcConstraintArrayI * impl)
public	IlcConstraintArray(IloSolver solver, IlcInt size)

Method Summary	
public void	display(ostream & stream) const
public IlcConstraintArrayI *	getImpl() const
public IlcInt	getSize() const
public IloSolver	getSolver() const
public void	operator=(const IlcConstraintArray & array)
public IlcConstraint &	operator[](IlcInt rank) const

Constructors

public **IlcConstraintArray**()

This constructor creates a void array. In other words, the array is an empty handle with a null handle pointer. The elements of this array must then be assigned before it is used, exactly as when you declare any other pointer.

public **IlcConstraintArray**(IlcConstraintArrayI * impl)

This constructor creates a handle object (an instance of the class `IlcConstraintArray`) from a pointer to an object (an instance of the implementation class `IlcConstraintArrayI`).

`public` **`IlcConstraintArray`**`(IloSolver solver, IlcInt size)`

This constructor creates an array of `size` elements. The elements of this array are not initialized. The argument `size` must be strictly greater than 0 (zero). Each element of the array must be assigned before the array can be used.

Methods

`public void` **`display`**`(ostream & stream) const`

This member function is called by the operator `<<`.

`public IlcConstraintArrayI *` **`getImpl`**`() const`

This member function returns a pointer to the implementation object of the invoking array. You can use this member function to check whether an array is empty.

`public IlcInt` **`getSize`**`() const`

This member function returns the number of constraints in the invoking array.

`public IloSolver` **`getSolver`**`() const`

This member function returns an instance of `IloSolver` associated with the invoking array.

`public void` **`operator=`**`(const IlcConstraintArray & array)`

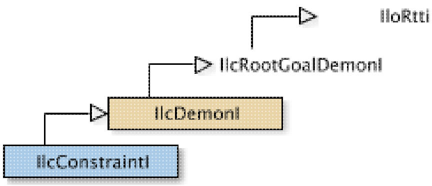
This assignment operator copies array into the invoking array by assigning an address to the handle pointer of the invoking object. That address is the location of the implementation object of the argument array.

`public IlcConstraint &` **`operator[]`**`(IlcInt rank) const`

This subscripting operator returns a reference to a constraint corresponding to the given index in the invoking array of constraints.

Class IlcConstraintI

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>



A constraint is an object in Solver. Like other Solver entities, a constraint is implemented by means of two classes: a handle class and an implementation class. In other words, an instance of the class `IlcConstraint` (a handle) contains a data member (the handle pointer) that points to an instance of the class `IlcConstraintI` (its implementation object).

Two member functions in this class actually implement the semantics of a constraint: `isViolated` and `propagate`. When you are defining a new class of constraint, you must define the member function `propagate`. That member function defines how the domain of a constrained variable must be reduced by the constraint. (Defining `isViolated` is not usually mandatory. Its definition is mandatory if you want to use the new class of constraint as a *metaconstraint*, for example in a Boolean formula.)

A constraint must be stored when it is posted so that it can be used by the propagation algorithm later. The member function `post` must be defined for that purpose. In other words, if you are defining a new class of constraint, when you define the implementation class for it, you must define this `post` member function as well.

A constraint can be used in a Boolean formula. In other words, you can combine constraints by means of the usual Boolean operators to produce other constraints. In order to use a constraint in that way, if you are defining a new class of constraints, then you must define the member functions `metaPost`, `makeOpposite`, and `isViolated`.

For more information, see the concepts Propagation and Propagation Events.

See Also: IlcConstraint, ILCCTDEMON0, IlcDemonI, IlcGoalI

Constructor and Destructor Summary	
`public`	`IlcConstraintI(IloSolver solver)`

Method Summary	
`public virtual void`	`display(ostream & str) const`
`public void`	`fail(IlcAny label=0)`
`public IlcConstraintI *`	`getCopy(IlcManagerI *=0)`
`public IlcDemonI *`	`getParentDemonI() const`
`public virtual IlcBool`	`isAConstraint() const`
`public virtual IlcBool`	`isViolated() const`
`protected virtual IlcConstraintI *`	`makeCopy(IlcManagerI *) const`
`public virtual IlcConstraintI *`	`makeOpposite() const`
`public virtual void`	`metaPostDemon(IlcDemonI * metaconstraint)`
`public virtual void`	`post()`
`public virtual void`	`propagate()`
`public void`	`push()`

public void	push(IlcInt priority)

Inherited Methods from **IlcDemonI**
getConstraintI, getSolver, getSolverI, isAConstraint, propagate

Constructors and Destructors

public **IlcConstraintI**(IloSolver solver)

This constructor creates a constraint implementation. This constructor should *not* be called directly because this is an *abstract* class. This constructor is called automatically in the constructors of its subclasses.

Methods

public virtual void **display**(ostream & str) const

By default, this virtual member function puts the name of the invoking constraint (if it has a name) on the output stream indicated by its argument; if the invoking constraint has no name, this virtual member function puts the string IlcConstraintI on that stream. When you define a new class of constraint, of course, you can redefine this behavior.

public void **fail**(IlcAny label=0)

This member function causes the invoking constraint to fail at the choice point indicated by label.

public IlcConstraintI * **getCopy**(IlcManagerI *=0)

This member function returns a pointer to a copy of the invoking implementation object.

public IlcDemonI * **getParentDemonI**() const

If the invoking constraint is defined at the top level (that is, it is not nested; it is inside another constraint), then this member function returns the constraint itself. If the invoking constraint is inside another constraint or inside a demon, then this member function returns that other constraint or demon.

public virtual IlcBool **isAConstraint**() const

This member function returns IlcTrue if the invoking constraint derives from IlcConstraintI; it returns IlcFalse otherwise.

public virtual IlcBool **isViolated**() const

If this member function returns IlcTrue, the invoking constraint cannot be satisfied. This member function may return IlcFalse even if the constraint can *not* be satisfied. However, it should *never* return IlcTrue if there is a possibility of satisfying the constraint. This provision is made for cases where it can be computationally expensive to determine whether the constraint can be satisfied or not; in such a case, the function should return IlcFalse. Consistent with this remark, the default behavior defined in the class IlcConstraintI for this member function is to return `IlcFalse`.

Since this virtual member function implements part of the semantics of an invoking constraint, it is not mandatory to redefine it in all cases when you define a new class of constraint. It is mandatory if you want to use instances of the new class in Boolean expressions, as explained about logical Boolean operators for the class `IlcConstraint`.

`public virtual IlcConstraintI *` **`makeOpposite`**`() const`

The negation of a constraint must also be a constraint. Semantically, this virtual member function expresses that principle. It is called to create the negation of the invoking constraint. This member function is called only once by Solver. Solver stores its results in order to avoid unnecessary computations.

This virtual member function must be defined when you define a new class of constraint if you want to use instances of the constraint in Boolean expressions, as explained about logical Boolean operators for the class `IlcConstraint`.

`public virtual void` **`metaPostDemon`**`(IlcDemonI * metaconstraint)`

When a Boolean expression is posted on constraints, the expression has to be examined whenever the truth value of one of the constraints appearing in it changes. That is, a constraint is posted on the constraints appearing in the expression. Since it is a constraint on constraints, we call it a *metaconstraint*. The virtual member function `metaPostDemon` must be defined when you are defining a new class of constraints if you plan to use instances of the new class in Boolean expressions, as explained about logical Boolean operators for the class `IlcConstraint`.

This virtual member function is called to post its argument, a metaconstraint, on the invoking constraint. The metaconstraint should be associated with all the propagation events of the expressions appearing in the invoking constraint that may result in the unsatisfiability of the invoking constraint. Normally, these propagation events are the same events that are used in the `post` member function.

Thus the implementation of `metaPostDemon` is very similar to the implementation of `post`. The main difference is that `metaPostDemon` associates propagation events with the *demon* passed as an argument, whereas `post` associates propagation events with the invoking *constraint*.

`public virtual void` **`post`**`()`

A constraint must be stored when it is posted so that it can be used by the propagation algorithm later. This member function must be defined for that purpose. In other words, if you are defining a new class of constraint, when you define the implementation class for it, you must define this pure virtual member function. It is called to attach the invoking constraint to the constrained expressions that the constraint involves.

This member function must associate the invoking constraint with propagation events triggered by the expressions it is constraining. This association is carried out by the member functions `whenValue`, `whenDomain`, `whenRange` (member functions of the classes `IlcIntExp`, `IlcAnyExp`, `IlcFloatExp`, `IlcIntSetVar`, `IlcAnySetVar`, etc.).

We strongly recommend that you do *not* modify variables in the scope of this member function. In other words, you should *not* call a modifier on a constrained variable within your definition of this virtual member function. Instead, you should make such changes (if they are necessary in your application) in your definition of the `propagate` member function.

`public virtual void` **`propagate`**`()`

This pure virtual member function must be redefined when you define a new class of constraints. It defines how the domains of constrained variables must be reduced by the invoking constraint. It is called by the constraint propagation algorithm in order to execute the propagation of the invoking constraint. This member function should reduce the domains of the constrained expressions involved in the invoking constraint by removing the values that cannot satisfy the invoking constraint.

While there is an active demon, you must *not* start another search in the same solver (the instance of `IloSolver` in which the invoking constraint exists). In practice, this rule means that you should *not* call `IloSolver::solve` from inside the member function `IlcConstraintI::propagate`.

`public void` **`push`**`()`

This member function pushes the invoking constraint onto the constraint priority queue. Solver will use the default priority in the queue.

`public void` **`push`**`(IlcInt priority)`

This member function pushes the invoking constraint onto the constraint priority queue. The argument `priority` indicates its priority in the queue.

`protected virtual IlcConstraintI *` **`makeCopy`**`(IlcManagerI *) const`

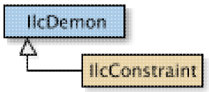
This virtual member function returns a pointer to a copy of the invoking implementation object and associates that copy with `solver`. When you derive a new class of constraints, you must, of course, define this virtual member function appropriately. In particular, you must insure that the copy of the constraint is built with a copy of each constrained variable involved in the constraint. In other words, `makeCopy` should copy the subobjects of the constraint; to do so, it should use the member function `getCopy`.

For example, if we define an equality constraint between two constrained integer variables `_x` and `_y`, we should implement `makeCopy` for that constraint like this:

```
 IlcConstraintI* MyEqCt::makeCopy(IloSolverI* solver) const {
        return new (solver) MyEqCt(_x,getCopy(solver),
                                   _y.getCopy(solver));
   }
```

Class IlcDemon

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>



Instances of `IlcDemon` are handles to *demons*. Demons differ from goals in these ways:

- Demons are executed immediately.
- You must associate a constraint with a demon. The member function `IlcDemon::getConstraint` returns the constraint associated with a demon.

For more information, see the concepts Propagation and Propagation Events.

See Also: ILCCTDEMON0, IlcDemonI, IlcTraceI

Constructor Summary	
`public`	`IlcDemon()`
`public`	`IlcDemon(IlcDemonI * impl)`
`public`	`IlcDemon(const IlcDemon & demon)`

Method Summary	
`public IlcConstraint`	`getConstraint() const`
`public IlcDemonI *`	`getImpl() const`
`public IloSolver`	`getSolver() const`
`public void`	`operator=(const IlcDemon & h)`

Constructors

`public` **`IlcDemon`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcDemon`**`(IlcDemonI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public` **`IlcDemon`**`(const IlcDemon & demon)`

This constructor creates a handle demon from the reference indicated by `demon`.

Methods

`public IlcConstraint` **`getConstraint`**`() const`

This member function returns a handle of the constraint associated with the invoking demon. If there is no such constraint, this member function returns 0 (zero). The invoking demon is known as the *parent* of the constraint.

```
public IlcDemonI * getImpl() const
```

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloSolver getSolver() const
```

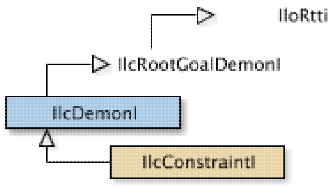
This member function returns an instance of `IloSolver` associated with the invoking object.

```
public void operator=(const IlcDemon & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IlcDemonI

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>



Instances of `IlcDemonI` are called *demons*. Demons differ from goals in these ways:

- Demons are executed immediately.
- You must associate a constraint with a demon. The member function `IlcDemonI::getConstraintI` returns the constraint associated with a demon.

For more information, see the concepts Propagation and Propagation Events.

See Also: IlcConstraintI, IlcGoalI, ILCCTDEMON0

Constructor and Destructor Summary	
`public`	`IlcDemonI(IloSolver solver, IlcConstraintI * ct=0)`

Method Summary	
`public IlcConstraintI *`	`getConstraintI() const`
`public IloSolver`	`getSolver() const`
`public IloSolverI *`	`getSolverI() const`
`public virtual IlcBool`	`isAConstraint() const`
`public virtual void`	`propagate()`

Constructors and Destructors

`public` **`IlcDemonI`**`(IloSolver solver, IlcConstraintI * ct=0)`

This constructor creates a demon implementation. This constructor should *not* be called directly because this is an *abstract* class. This constructor is called automatically in the constructors of its subclasses. The constraint passed as a parameter is the constraint constraint associated with this demon.

Methods

`public IlcConstraintI *` **`getConstraintI`**`() const`

This member function returns a pointer to the implementation of the constraint associated with the invoking demon. The demon is known as the parent of the constraint.

`public IloSolver` **`getSolver`**`() const`

This member function returns the solver (a handle) of the invoking demon implementation.

`public IloSolverI * getSolverI() const`

This member function returns a pointer to the implementation object of the solver where the invoking demon was extracted.

`public virtual IlcBool isAConstraint() const`

This member function returns `IlcTrue` if the invoking demon is an instance of `IlcConstraintI`; it returns `IlcFalse` otherwise. In other words, it lets you know whether the invoking demon is a constraint or a demon.

`public virtual void propagate()`

This member function propagates the invoking demon. Normally, an implementation of this virtual member function will call a member function of the associated constraint.

Class IlcFloatArray

Definition file: ilsolver/fltexp.h
Include file: <ilsolver/ilosolver.h>

For each basic type, Solver defines a corresponding array class. This array class is a handle class. In other words, an object of this class contains a pointer to another object allocated on the Solver heap associated with a solver (an instance of `IloSolver`). Exploiting handles in this way greatly simplifies the programming interface since the handle can then be an automatic object: as a developer using handles, you do not have to worry about memory allocation.

`IlcFloatArray` is the array class for the basic floating-point class. It is a handle class. The implementation class for `IlcFloatArray` is the undocumented class `IlcFloatArrayI`.

Empty Handle or Null Array

It is possible to create a null array, or in other words, an empty handle. When you do so, only these operations are allowed on that null array:

- **copy**: You can assign the null array to a new array.
- **access its size**: The member function `getSize` for the null array returns 0 (zero).
- **create an iterator**: You can create an iterator to traverse the null array. The member function `ok` returns `IlcFalse` for a null array.

Attempts to access a null array in any other way will throw an exception (an instance of `IloSolver::SolverErrorException`).

See Also: IlcConstFloatArray, IlcFloat, operator<<

Constructor Summary	
`public`	`IlcFloatArray()`
`public`	`IlcFloatArray(IlcFloatArrayI * impl)`
`public`	`IlcFloatArray(IloSolver s, IlcInt size, IlcFloat * values)`
`public`	`IlcFloatArray(IloSolver solver, IlcInt size, IlcFloat prototype=0)`
`public`	`IlcFloatArray(IloSolver solver, IlcInt size, IlcFloat exp0, IlcFloat exp1, ...)`
`public`	`IlcFloatArray(IloSolver solver, IlcInt size, IlcInt exp0, IlcInt exp1, ...)`

Method Summary	
`public IlcFloatArrayI *`	`getImpl() const`
`public IlcInt`	`getSize() const`
`public IloSolver`	`getSolver() const`
`public void`	`operator=(const IlcFloatArray & h)`
`public IlcFloat &`	`operator[](IlcInt i) const`

Constructors

`public` **`IlcFloatArray`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public **IlcFloatArray**(IlcFloatArrayI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public **IlcFloatArray**(IloSolver s, IlcInt size, IlcFloat * values)`

This constructor creates an array of floating-point numbers containing the values in the array `values`. The argument `size` must be the length of the array `values`. It must also be strictly greater than 0 (zero). Solver does not keep a pointer to the array `values`. When you create an array of floating-point values, the elements of the array must be of the same type (for example, all floating-point, or all integer, but not a mixture of the two) because those types are not necessarily the same size in C++. You can write this:

```
IlcFloatArray arrayok (s, 3, 1., 3., 2.);
```

or this:

```
IlcFloatArray arrayOK(s, 3, 1, 3, 2);
```

but not this:

```
IlcFloatArray notok(s, 3, 1., 3, 2.); // bad idea
```

in which some values are floating-point, some are integer, and consequently of different sizes in C++.

`public **IlcFloatArray**(IloSolver solver, IlcInt size, IlcFloat prototype=0)`

This constructor creates an array of `size` elements. The argument `size` must be strictly greater than 0 (zero). The elements of this array are not initialized.

`public **IlcFloatArray**(IloSolver solver, IlcInt size, IlcFloat exp0, IlcFloat exp1, ...)`

This constructor accepts a variable number of arguments. Its first argument, `size`, indicates the length of the array that this constructor will create; `size` must be the number of arguments minus one (that is, the number of arguments of type `IlcFloat`); it must also be strictly greater than 0 (zero). The constructor creates an array of the values indicated by the other arguments. The arguments, `exp0`, `exp1`, etc. are all of the same type. Do not mix types within an array.

`public **IlcFloatArray**(IloSolver solver, IlcInt size, IlcInt exp0, IlcInt exp1, ...)`

This constructor accepts a variable number of arguments. Its first argument, `size`, indicates the length of the array that this constructor will create; `size` must be the number of arguments minus one (that is, the number of arguments of type `IlcInt`); it must also be strictly greater than 0 (zero). The constructor creates an array of the values indicated by the other arguments. The arguments, `exp0`, `exp1`, etc. are all of the same type. Do not mix types within an array.

Methods

`public IlcFloatArrayI * **getImpl**() const`

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

`public IlcInt **getSize**() const`

This member function returns the number of elements in the invoking array.

`public IloSolver **getSolver**() const`

This member function returns an instance of `IloSolver` associated with the invoking object.

`public void **operator=**(const IlcFloatArray & h)`

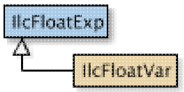
This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

`public IlcFloat & **operator[]**(IlcInt i) const`

This operator returns a reference to the element at rank `i`. This operator can be used for accessing (that is, simply reading) the element or for modifying (that is, writing) it.

Class IlcFloatExp

Definition file: ilsolver/fltexp.h
Include file: <ilsolver/ilosolver.h>



The class `IlcFloatExp` is the root for constrained floating-point expressions. A constrained floating-point expression computes its domain from the domains of its subexpressions.

In fact, a constrained floating-point *variable* itself is a constrained floating-point *expression*: the class `IlcFloatVar` is a subclass of `IlcFloatExp`, so a constrained floating-point variable is simply a constrained floating-point expression whose domain is stored. Since the number of elements in the domain of a constrained continuous floating-point variable is very high (typically millions), there is usually an *interval* associated with the variable to represent its domain. The domain of a constrained discrete floating-point variable (that is, one constructed by `IlcFloatVar`) can be explicitly enumerated so its domain is represented by an array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Precision

In Solver, all computations on floating-point values are computed in *doubleprecision* mode, that is, 64 bits on most hardware.

A *relativeprecision* is associated with each constrained floating-point expression in Solver. This relative precision is taken into account during constraint propagation: if the size of the interval associated with a floating-point expression is less than the precision associated with that expression, then we consider the expression as bound to the mean value in that interval. In other words, the expression is bound to the average value in a such an interval. Consequently, in this context, precision represents a degree of uncertainty about this value. More formally, we say that a constrained floating-point variable `x` with a precision indicated by `precision` is *bound* when its associated interval is bounded by `min` and `max` such that `((max - min)/(max{1, |min|}) <= precision`.

The smaller the precision of a variable, the more precise are computations with it, but such computations can take more time, of course.

To compare a constrained floating-point expression to 0 (zero), use the function `IlcNull`.

Domain

The domain of a constrained floating-point expression (an instance of `IlcFloatExp`) is *computed* from the domains of its subexpressions. For example, if `x` and `y` are both instances of `IlcFloatExp`, then the domain of `x+y` contains the range `[x.getMin()+y.getMin(), x.getMax()+y.getMax()]`. (In contrast, the domain of a constrained floating-point variable (an instance of `IlcFloatVar`) is stored.) The domain of a constrained floating-point expression can be reduced to the point of being empty. In such a case, *failure* is triggered by means of the function `IlcFail` since no solution is then possible.

Checking Overflow and Underflow

Solver explicitly checks *floating-point* computations for overflow and underflow.

IEEE 754 is a standard proposed by the Institute of Electronic and Electrical Engineers for computing floating-point arithmetic. The implementation of floating-point numbers in Solver conforms to this standard. See the Solver User's Manual for a discussion of floating-point arithmetic.

Backtracking and Reversibility

All the member functions and operators defined for this class and capable of modifying constrained variables are *reversible*. In particular, the changes made by constraint-posting functions are made with reversible assignments. Thus, the value, the domain, and the constraints posted on any constrained variable are restored when Solver backtracks.

For more information, see the concept Propagation.

A modifier is a member function that reduces the domain of a constrained floating-point expression, if it can. The modifier is not stored, in contrast to a constraint. If the constrained floating-point expression is a constrained floating-point variable, the modifications due to the modifier call are stored in its domain. Otherwise, the effect of the modifier is propagated to the subexpressions of the constrained floating-point expression. If the domain becomes empty, failure occurs. Modifiers are usually used to define new *classes* of constraints.

See Also: IlcAbs, IlcCos, IlcExponent, IlcFloatExpIterator, IlcFloatSet, IlcFloatVar, IlcFloatVarDeltaIterator, IlcLog, IlcMax, IlcMin, IlcMonotonicDecreasingFloatExp, IlcMonotonicIncreasingFloatExp, IlcNull, IlcPower, IlcScalProd, IlcSin, IlcSolveBounds, IlcSquare, IlcSum, IlcTan, operator+, operator-, operator*, operator/, operator<<

Constructor Summary	
public	IlcFloatExp()
public	IlcFloatExp(IlcFloatExpI * impl)
public	IlcFloatExp(IlcIntExp exp)

Method Summary	
public void	display(ostream & str) const
public IlcFloatExp	getCopy() const
public IlcFloatExpI *	getImpl() const
public IlcFloat	getMax() const
public IlcFloat	getMin() const
public const char *	getName() const
public IlcFloat	getNextHigher(IlcFloat val) const
public IlcFloat	getNextLower(IlcFloat val) const
public IlcAny	getObject() const
public IlcFloat	getPrecision() const
public IlcFloat	getSize() const
public IloSolver	getSolver() const
public IloSolverI *	getSolverI() const
public IlcBool	isBound() const
public IlcBool	isInDomain(IlcFloat value) const
public void	operator=(const IlcFloatExp & h)
public void	removeDomain(IlcFloatSet domain)
public void	removeDomain(IlcFloatArray domain)
public void	removeRange(IlcFloat min, IlcFloat max) const
public void	removeValue(IlcFloat val) const
public void	setDomain(IlcFloatArray domain)
public void	setDomain(IlcFloatSet domain)
public void	setDomain(IlcFloatExp var)

public void	setMax(IlcFloat max) const
public void	setMin(IlcFloat min) const
public void	setName(const char * name) const
public void	setObject(IlcAny object) const
public void	setPrecision(IlcFloat precision)
public void	setRange(IlcFloat min, IlcFloat max) const
public void	whenDomain(const IlcDemon ct) const
public void	whenRange(const IlcDemon demon) const
public void	whenValue(const IlcDemon demon) const

Constructors

public **IlcFloatExp**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IlcFloatExp**(IlcFloatExpI * impl)

This constructor creates a handle object from a pointer to an implementation object.

public **IlcFloatExp**(IlcIntExp exp)

A constrained integer expression in Solver can be seen as a constrained floating-point expression since integers can be converted to floating-point values. This constructor is for casting a constrained integer expression into a constrained floating-point expression. This constructor creates a constrained floating-point expression which is constrained to be equal to the constrained integer expression `exp`. In other words, you can use this constructor to define a constrained floating-point expression to be equal to a constrained integer expression. Such a floating-point expression can then be used like any other constrained floating-point expression. It is thus possible to combine integers and floating-point expressions within a constraint.

Usually, casting a constrained integer expression to a constrained floating-point expression is done automatically by the compiler, so you don't ordinarily need to use this constructor. In fact, you should use it only if your compiler warns you that something is wrong when you combine constrained floating-point and integer expressions.

Methods

public void **display**(ostream & str) const

This member function puts the invoking object on the output stream indicated by its argument.

public IlcFloatExp **getCopy**() const

This member function returns a copy of invoking constrained floating-point expression and associates that copy with `solver`.

public IlcFloatExpI * **getImpl**() const

This constructor creates an object by copying another one. This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

public IlcFloat **getMax**() const

This member function returns the maximum value of the invoking object.

public IlcFloat **getMin**() const

This member function returns the minimum value of the invoking object.

public const char * **getName**() const

This member function returns the name of the invoking object.

public IlcFloat **getNextHigher**(IlcFloat val) const

This member function applies only to discrete floating-point expressions (that is, those created with an explicitly enumerated domain by the constructor `IlcFloatVar`). This member function returns the least floating-point value that is greater than or equal to `val` from the domain of the invoking discrete floating-point expression.

If you apply this member function to a continuous floating-point expression (that is, an instance of `IlcFloatExp` not constructed by `IlcFloatVar`), it will throw an exception (an instance of `IloSolver::SolverErrorException`).

public IlcFloat **getNextLower**(IlcFloat val) const

This member function applies only to discrete floating-point expressions (that is, those created with an explicitly enumerated domain by the constructor `IlcFloatVar`). This member function returns the greatest floating-point value that is less than or equal to `val` from the domain of the invoking discrete floating-point expression.

If you apply this member function to a continuous floating-point expression (that is, an instance of `IlcFloatExp` not constructed by `IlcFloatVar`), it will throw an exception (an instance of `IloSolver::SolverErrorException`).

public IlcAny **getObject**() const

This member function returns a pointer to the external object associated with the invoking object, if there is such an association. It returns 0 (zero) otherwise.

public IlcFloat **getPrecision**() const

This member function returns the precision of the invoking constrained floating-point expression. We say that a constrained floating-point variable `x` with a precision indicated by `precision` is *bound* when its associated interval is bounded by `min` and `max` such that $\frac{max - min}{max(1,|min|)} \leq precision$

The smaller the precision of a variable, the more precise are computations with it, but such computations can take more time, of course.

public IlcFloat **getSize**() const

This member function returns the width of the domain of the invoking constrained floating-point expression. By width of the domain, we mean the difference between the two boundaries of the domain.

`public IloSolver getSolver() const`

This member function returns an instance of `IloSolver` associated with the invoking object.

`public IloSolverI * getSolverI() const`

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

`public IlcBool isBound() const`

This member function returns `IlcTrue` if the invoking constrained floating-point expression is bound. Otherwise, the member function returns `IlcFalse`.

`public IlcBool isInDomain(IlcFloat value) const`

This member function returns `IlcTrue` if `value` is in the domain of the invoking constrained floating-point expression. Otherwise, the member function returns `IlcFalse`.

`public void operator=(const IlcFloatExp & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

`public void removeDomain(IlcFloatSet domain)`

This member function applies only to discrete floating-point expressions (that is, those created with an explicitly enumerated domain by the constructor `IlcFloatVar`). This member function removes the values indicated by the set `domain` from the domain of the invoking discrete floating-point expression.

If you apply this member function to a continuous floating-point expression (that is, an instance of `IlcFloatExp` not constructed by `IlcFloatVar`), it will throw an exception (an instance of `IloSolver::SolverErrorException`).

`public void removeDomain(IlcFloatArray domain)`

This member function applies only to discrete floating-point expressions (that is, those created with an explicitly enumerated domain by the constructor `IlcFloatVar`). This member function removes the values indicated by the array `domain` from the domain of the invoking discrete floating-point expression.

If you apply this member function to a continuous floating-point expression (that is, an instance of `IlcFloatExp` not constructed by `IlcFloatVar`), it will throw an exception (an instance of `IloSolver::SolverErrorException`).

`public void removeRange(IlcFloat min, IlcFloat max) const`

This member function applies only to discrete floating-point expressions (that is, those created with an explicitly enumerated domain by the constructor `IlcFloatVar`). This member function removes all the values between `min` and `max`, inclusive, from the domain of the invoking discrete floating-point expression.

If you apply this member function to a continuous floating-point expression (that is, an instance of `IlcFloatExp` not constructed by `IlcFloatVar`), it will throw an exception (an instance of `IloSolver::SolverErrorException`).

`public void **removeValue**(IlcFloat val) const`

This member function applies only to discrete floating-point expressions (that is, those created with an explicitly enumerated domain by the constructor `IlcFloatVar`). This member function removes `val` from the domain of the invoking discrete floating-point expression. If `val` was not in the domain of the invoking discrete expression, then this member function does nothing.

If you apply this member function to a continuous floating-point expression (that is, an instance of `IlcFloatExp` not constructed by `IlcFloatVar`), it will throw an exception (an instance of `IloSolver::SolverErrorException`).

When it removes a value from a domain, Solver may need to allocate more memory for the representation of the remaining domain. The amount of memory allocated depends on the size of the domain of the variable.

`public void **setDomain**(IlcFloatArray domain)`

This member function applies only to discrete floating-point expressions (that is, those created with an explicitly enumerated domain by the constructor `IlcFloatVar`). This member function assigns `domain` as the domain of the invoking discrete floating-point expression.

If you apply this member function to a continuous floating-point expression (that is, an instance of `IlcFloatExp` not constructed by `IlcFloatVar`), it will throw an exception (an instance of `IloSolver::SolverErrorException`).

`public void **setDomain**(IlcFloatSet domain)`

This member function applies only to discrete floating-point expressions (that is, those created with an explicitly enumerated domain by the constructor `IlcFloatVar`). This member function assigns `domain` as the domain of the invoking discrete floating-point expression.

If you apply this member function to a continuous floating-point expression (that is, an instance of `IlcFloatExp` not constructed by `IlcFloatVar`), it will throw an exception (an instance of `IloSolver::SolverErrorException`).

`public void **setDomain**(IlcFloatExp var)`

This member function applies only to discrete floating-point expressions (that is, those created with an explicitly enumerated domain by the constructor `IlcFloatVar`). This member function assigns `var` as the domain of the invoking discrete floating-point expression.

If you apply this member function to a continuous floating-point expression (that is, an instance of `IlcFloatExp` not constructed by `IlcFloatVar`), it will throw an exception (an instance of `IloSolver::SolverErrorException`).

`public void **setMax**(IlcFloat max) const`

This member function removes all the elements that are greater than `max` from the domain of the invoking constrained expression. If the domain thus becomes empty, then the function `IlcFail` is called. Otherwise, if the domain is modified, the propagation event `range` is generated. Moreover, if the invoking constrained floating-point expression becomes bound, then the propagation event `value` is also generated. The effects of this member function are reversible.

`public void` **`setMin`**`(IlcFloat min) const`

This member function removes all the elements that are less than `min` from the domain of the invoking constrained expression. If the domain thus becomes empty, then t he function `IlcFail` is called. Otherwise, if the domain is modified, the propagation event `range` is generated. Moreover, if the invoking constrained floating-point expression becomes bound, then the propagation event `value` is also generated. The effects of this member function are reversible.

`public void` **`setName`**`(const char * name) const`

This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

`public void` **`setObject`**`(IlcAny object) const`

This member function establishes a link between the invoking object and an external object of which the invoking object might be a data member.

`public void` **`setPrecision`**`(IlcFloat precision)`

This member function sets the precision of the invoking constrained floating-point expression. This is a non-reversible action.

`public void` **`setRange`**`(IlcFloat min, IlcFloat max) const`

This member function removes all the elements that are either less than `min` or greater than `max` from the domain of the invoking constrained expression. If the domain thus becomes empty, then the function `IlcFail` is called. Otherwise, if the domain is modified, the propagation event `range` is generated. Moreover, if the invoking constrained floating-point expression becomes bound, then the propagation event `value` is also generated. The effects of this member function are reversible.

`public void` **`whenDomain`**`(const IlcDemon ct) const`

This member function applies only to discrete floating-point expressions (that is, those created with an explicitly enumerated domain by the constructor `IlcFloatVar`). This member function associates the demon `ct` with the domain event of the invoking discrete floating-point expression. When the domain of the discrete expression changes, the demon executes immediately.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. Whenever the domain of the invoking constrained expression changes, the constraint will be propagated.

If you apply this member function to a continuous floating-point expression (that is, an instance of `IlcFloatExp` not constructed by `IlcFloatVar`), it will throw an exception (an instance of `IloSolver::SolverErrorException`).

`public void` **`whenRange`**`(const IlcDemon demon) const`

This member function associates `demon` with the range event of the invoking constrained expression. Whenever one of the boundaries of the domain of the invoking constrained expression changes, the demon will be executed immediately.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. Whenever one of the boundaries of the domain of the invoking constrained expression changes, the constraint will be propagated.

public void **whenValue**(const IlcDemon demon) const

This member function associates `demon` with the value event of the invoking constrained expression. Whenever the invoking constrained expression becomes bound, the demon will be executed immediately.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. Whenever the invoking constrained expression becomes bound, the constraint will be propagated.

Class IlcFloatExpIterator

Definition file: ilsolver/fltexp.h
Include file: <ilsolver/ilosolver.h>

An instance of the class `IlcFloatExpIterator` is an iterator that traverses the values belonging to the domain of a constrained discrete floating-point expression. A discrete floating-point variable is an instance of `IlcFloatVar` defined by this constructor:

```
IlcFloatVar
```

In other words, it is a floating-point variable with an enumerated domain.

An iterator of this class will not traverse the domain of a continuous floating-point variable (that is, one created by the other constructors of `IlcFloatVar`) because its domain is represented by an interval (not an enumerated set).

For more information, see the concept Iterator.

See Also: IlcFloatExp, IlcFloatVar

Constructor and Destructor Summary	
public	IlcFloatExpIterator(IlcFloatExp exp)

Method Summary	
public IlcBool	ok() const
public IlcFloat	operator*() const
public IlcFloatExpIterator &	operator++()

Constructors and Destructors

public **IlcFloatExpIterator**(IlcFloatExp exp)

This constructor creates an iterator associated with `exp` to traverse the values belonging to the domain of `exp`. `exp` must be a discrete floating-point expression.

Methods

public IlcBool **ok**() const

This member function returns `IlcTrue` if there is a current element and the iterator points to it. Otherwise, it returns `IlcFalse`.

To traverse the values belonging to the domain of a constrained discrete floating-point expression, use the following code:

```
IlcFloat val;
for (IlcFloatExpIterator iter(exp); iter.ok(); ++iter){
        val = *iter;
        // do something with val
}
```

`public IlcFloat` **`operator*`**`() const`

This operator returns the current element, the one to which the invoking iterator points.

`public IlcFloatExpIterator &` **`operator++`**`()`

This operator advances the iterator to point to the next value in the domain of the constrained discrete floating-point expression.

Class IlcFloatSet

Definition file: ilsolver/fltexp.h
Include file: <ilsolver/ilosolver.h>

Finite sets of discrete, enumerated floating-point numbers are instances of the handle class `IlcFloatSet`. These sets are used by Solver to represent the domains of enumerated constrained variables and to represent the values of constrained set variables. Solver provides an efficient, optimized implementation of finite sets of floating-point numbers. These finite sets are known formally as instances of the handle class `IlcFloatSet`.

The elements of finite sets of type `IlcFloatSet` are floating-point numbers of type `IlcFloat`. The implementation class for finite sets of floating-point numbers is the undocumented class `IlcFloatSetI`.

To traverse an existing finite set, either exhaustively or in search of an element, Solver provides iterators (such as instances of `IlcFloatSetIterator`). An iterator is an object constructed from a data structure (such as a set or an array) and contains a traversal state of this data structure.

See Also: IlcFloat, IlcFloatExp, IlcFloatExpIterator, IlcFloatSetIterator, IlcFloatVar, IlcFloatVarDeltaIterator

Constructor Summary	
`public`	`IlcFloatSet()`
`public`	`IlcFloatSet(IlcFloatSetI * impl)`
`public`	`IlcFloatSet(const IlcFloatArray array, IlcBool fullSet=IlcTrue)`
`public`	`IlcFloatSet(IloSolver solver, const IlcFloatArray array, IlcBool fullSet=IlcTrue)`

Method Summary	
`public IlcBool`	`add(IlcFloat elt)`
`public IlcFloatSet`	`copy() const`
`public IlcFloatSetI *`	`getImpl() const`
`public IlcInt`	`getSize() const`
`public IloSolver`	`getSolver() const`
`public IlcBool`	`isIn(IlcFloat elt) const`
`public void`	`operator=(const IlcFloatSet & h)`
`public IlcBool`	`remove(IlcFloat elt)`

Constructors

`public` **`IlcFloatSet`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcFloatSet`**`(IlcFloatSetI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public IlcFloatSet(const IlcFloatArray array, IlcBool fullSet=IlcTrue)`

This constructor also creates a finite set of floating-point numbers containing the elements of `array`.

`public IlcFloatSet(IloSolver solver, const IlcFloatArray array, IlcBool fullSet=IlcTrue)`

This constructor creates a finite set of floating-point numbers containing the elements of `array`. If `array` contains multiple copies of a given value, that value will appear only one time in the newly created finite set. If the argument `fullSet` is equal to `IlcTrue`, its default value, the finite set will initially contain all its possible values. Otherwise, the finite set will initially be empty. In any case, the possible elements of the finite set are exactly those elements in `array`.

Methods

`public IlcBool add(IlcFloat elt)`

This member function adds `elt` to the invoking finite set if `elt` is a possible member of that set and if `elt` is not already in that set. When both conditions are met, this member function returns `IlcTrue`. Otherwise, it returns `IlcFalse`. The effects of this member function are reversible.

`public IlcFloatSet copy() const`

This member function creates and returns a finite set that contains the same elements as the invoking finite set.

`public IlcFloatSetI * getImpl() const`

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

`public IlcInt getSize() const`

This member function returns the size of a finite set. Clearly, this member function is useful for testing whether the invoking finite set is empty or not.

`public IloSolver getSolver() const`

This member function returns an instance of `IloSolver` associated with the invoking object.

`public IlcBool isIn(IlcFloat elt) const`

This member function is a predicate that indicates whether or not `elt` is in the invoking finite set. It returns `IlcTrue` if `elt` is in the set; otherwise, it returns `IlcFalse`.

```
public void operator=(const IlcFloatSet & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

```
public IlcBool remove(IlcFloat elt)
```

This member function removes `elt` from the invoking finite set. This member function returns `IlcTrue` if `elt` was not in that invoking set; otherwise, it returns `IlcFalse`. The effects of this member function are reversible.

Class IlcFloatSetIterator

Definition file: ilsolver/fltexp.h
Include file: <ilsolver/ilosolver.h>

An instance of the class `IlcFloatSetIterator` is an iterator that traverses the elements of a finite set of discrete floating-point numbers (instance of `IlcFloatSet`).

For more information, see the concept Iterator.

See Also: IlcFloatSet

Constructor Summary	
`public`	`IlcFloatSetIterator(IlcFloatSet set)`

Method Summary	
`public IlcBool`	`ok() const`
`public IlcFloat`	`operator*() const`
`public IlcFloatSetIterator &`	`operator++()`

Constructors

`public` **`IlcFloatSetIterator`**`(IlcFloatSet set)`

This constructor creates an iterator associated with `set` to traverse its elements.

Methods

`public IlcBool` **`ok`**`() const`

This member function returns `IlcTrue` if there is a current element and the invoking iterator points to it. Otherwise, it returns `IlcFalse`.

To traverse the elements of a finite set of discrete floating-point numbers, use the following code:

```
IlcAny val;
for(IlcFloatSetIterator iter(set); iter.ok(); ++iter){
              val = *iter;
              // do something with val
}
```

`public IlcFloat` **`operator*`**`() const`

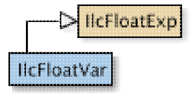
This operator returns the current element, the one to which the invoking iterator points.

`public IlcFloatSetIterator &` **`operator++`**`()`

This operator advances the iterator to point to the next value in the set.

Class IlcFloatVar

Definition file: ilsolver/fltexp.h
Include file: <ilsolver/ilosolver.h>



The class `IlcFloatVar` is a subclass of `IlcFloatExp`. A constrained floating-point variable (an instance of `IlcFloatVar`) is a constrained floating-point expression that stores its domain (instead of computing the domain from the domains of its subexpressions). The domain of a constrained floating-point variable contains values of type `IlcFloat`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Continuous and Discrete Enumerated Floating-Point Variables

Solver offers two different kinds of constrained floating-point variables: continuous and discrete or enumerated. Both kinds of variables assume values of type `IlcFloat`. They differ in the way their domains are represented, however.

A continuous floating-point variable is any instance of `IlcFloatExp` or `IlcFloatVar` not created by `IlcFloatVar` For continuous floating-point variables, the number of elements in the domain is very high (typically millions). Consequently, Solver associates an interval with the variable to represent its domain. Because it is not practical to count the elements in the domain of a continuous floating-point variable, there are no iterators for traversing the domain of continuous floating-point variables.

For users of Solver prior to version 5, these continuous variables defined on intervals were the only constrained floating-point variables available. Continuous floating-point variables are still available in Solver version 5; in addition, there are now also discrete or enumerated floating-point variables.

In contrast to continuous floating-point variables, discrete floating-point variables can be enumerated. That is, their domain can be counted explicitly. Consequently, Solver associates an enumerated set with the variable to represent its domain. To create a discrete floating-point variable, use the constructor

```
IlcFloatVar
```

With that constructor, you explicitly enumerate the values that the discrete variable may assume. This constructor offers the only way to create a discrete floating-point variable with an enumerated domain.

Failure

If the boundaries of the domain are identical and equal to minus infinity or plus infinity, then failure is triggered since no finite solution is then possible. That situation can happen with the following declaration, for example:

```
IloSolver s;
IlcFloatVar x(s,0,10);
IlcFloatVar y=IlcLog(x);
s.add( x == 0);
```

Those lines cause failure since the minimal and maximal boundaries of `y` are equal to minus infinity.

Domain-Delta And Propagation

When a propagation event is triggered for a constrained variable, the variable is pushed into the propagation queue if it was not already in the queue. Moreover, the modifications of the domain of the constrained variable are stored in a special set called the *domain-delta*. This domain-delta can be accessed during the propagation of the constraints posted on that variable. When all the constraints posted on that variable have been processed, then the domain-delta is cleared. If the variable is modified again, then the whole process begins again. The state of the domain-delta is reversible.

For discrete floating-point variables (that is, instances of `IlcFloatVar` created by the constructor `IlcFloatVar`) to traverse their domain-delta, Solver offers iterators of the class `IlcFloatVarDeltaIterator`.

Backtracking and Reversibility

All the member functions and operators defined for this class and capable of modifying constrained variables are *reversible*. In particular, the changes made by constraint-posting functions are made with reversible assignments. Thus, the value, the domain, and the constraints posted on any constrained variable are restored when Solver backtracks.

See Also: IlcFloat, IlcFloatExp, IlcFloatExpIterator, IlcFloatSet, IlcFloatVarArray, IlcFloatVarDeltaIterator, IlcSolveBounds

Constructor Summary	
public	IlcFloatVar()
public	IlcFloatVar(IloSolver solver, IlcFloat min, IlcFloat max, const char * name=0)
public	IlcFloatVar(IloSolver solver, IlcFloat min, IlcFloat max, IlcFloat precision, const char * name=0)
public	IlcFloatVar(IloSolver solver, const IlcFloatArray array, const char * name=0)
public	IlcFloatVar(IlcFloatVarI * impl)
public	IlcFloatVar(IlcIntVar var)
public	IlcFloatVar(const IlcFloatExp exp)

Method Summary	
public IlcFloat	getMax() const
public IlcFloat	getMaxDelta() const
public IlcFloat	getMin() const
public IlcFloat	getMinDelta() const
public IlcFloat	getOldMax() const
public IlcFloat	getOldMin() const
public IlcBool	isInDelta(IlcFloat value) const
public IlcBool	isInProcess() const
public void	operator=(const IlcFloatVar & exp)
public void	operator=(const IlcFloatExp & exp)

Inherited Methods from **`IlcFloatExp`**
display, getCopy, getImpl, getMax, getMin, getName, getNextHigher, getNextLower, getObject, getPrecision, getSize, getSolver, getSolverI, isBound, isInDomain, operator=, removeDomain, removeDomain, removeRange, removeValue, setDomain, setDomain, setDomain, setMax, setMin, setName, setObject, setPrecision, setRange, whenDomain, whenRange, whenValue

Constructors

`public IlcFloatVar()`

This constructor creates a constrained floating-point variable which is empty, that is, whose handle pointer is null. This object must then be assigned before it can be used, exactly as when you, as a developer, declare a pointer. To check whether a floating-point variable is empty, use the member function `IlcFloatExp::getImpl`.

`public IlcFloatVar(IloSolver solver, IlcFloat min, IlcFloat max, const char * name=0)`

This constructor creates a continuous constrained floating-point variable with a domain containing all the floating-point values between `min` and `max`, inclusive. If `min` is greater than `max`, the function `IlcFail` is called. The optional argument `name`, if provided, becomes the name of the constrained floating-point variable. The precision associated with the constrained floating-point variable will be the default precision of Solver.

`public IlcFloatVar(IloSolver solver, IlcFloat min, IlcFloat max, IlcFloat precision, const char * name=0)`

This constructor creates a continuous constrained floating-point variable with a domain containing all the floating-point values between `min` and `max`, inclusive. If `min` is greater than `max`, the function `IlcFail` is called. The argument `precision` becomes the precision associated with the constrained floating-point variable. If the optional argument `name` is provided, it becomes the name of the constrained floating-point variable.

For example, here's how to create a constrained floating-point variable with a minimum of 0, a maximum of 10, and a name.

```
IlcFloatVar x (s, 0, 10, "x");
```

Here's how to create a constrained floating-point variable with an associated precision of 10-4.

```
IlcFloatVar x (s, 0, 100, 1e-4);
```

You can specify both the name and the precision at the same time, like this:

```
IlcFloatVar x (s, -100, 100, 1e-4, "x");
```

`public IlcFloatVar(IloSolver solver, const IlcFloatArray array, const char * name=0)`

This constructor creates a *discrete* constrained floating-point variable. Its domain is enumerated by the array `array`.

`public IlcFloatVar(IlcFloatVarI * impl)`

This constructor creates a handle object (an instance of the class `IlcFloatVar` from a pointer to an object (an instance of the class `IlcFloatVarI`.

`public IlcFloatVar(IlcIntVar var)`

This constructor creates an instance of the class `IlcFloatVar`. This instance is constrained to be equal to the argument `var`.

`public` **`IlcFloatVar`**`(const IlcFloatExp exp)`

This constructor associates a domain with the continuous constrained floating-point expression `exp`. Moreover, the newly created floating-point variable points to the same implementation object as `exp`. In other words, this constructor transforms a constrained floating-point *expression* (which computes its domain from its subexpressions) into a constrained floating-point *variable* (which stores its domain).

Methods

`public IlcFloat` **`getMax`**`() const`

This member function returns the maximum of the domain of the invoking object.

`public IlcFloat` **`getMaxDelta`**`() const`

This member function returns the difference between the maximum of the domain of the invoking constrained variable and the maximum of its domain-delta. This member function can be applied only to the variable currently in process.

`public IlcFloat` **`getMin`**`() const`

This member function returns the minimum of the domain of the invoking object.

`public IlcFloat` **`getMinDelta`**`() const`

This member function returns the difference between the minimum of the domain of the invoking constrained variable and the minimum of its domain-delta. This member function can be applied only to the variable currently in process.

For example, to know whether the minimum of a constrained floating-point variable `x` has been modified since the last time the constraints posted on `x` were processed, it is sufficient to test the value of `x.getMinDelta()`. If that test returns 0, then the minimum of `x` has not been modified.

`public IlcFloat` **`getOldMax`**`() const`

This member function returns the maximum of the domain-delta of the invoking constrained variable. This member function can be applied only to the variable currently in process.

`public IlcFloat` **`getOldMin`**`() const`

This member function returns the minimum of the domain-delta of the invoking constrained variable. This member function can be applied only to the variable currently in process.

`public IlcBool` **`isInDelta`**`(IlcFloat value) const`

This member function returns `IlcTrue` if the argument `value` belongs to the domain-delta of the invoking constrained variable. This member function can be applied only to the variable currently in process.

`public IlcBool` **`isInProcess`**`() const`

This member function returns `IlcTrue` if the invoking constrained variable is currently being processed by the constraint propagation algorithm. Only one variable can be in process at a time.

`public void` **`operator=`**`(const IlcFloatVar & exp)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the argument `exp`. After the execution of this operator, the invoking object and the `exp` object both point to the same implementation object. This assignment operator has no effect on its argument.

`public void` **`operator=`**`(const IlcFloatExp & exp)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the argument `exp`. After the execution of this operator, the invoking object and `exp` both point to the same implementation object. Moreover, this assignment operator associates a domain with the constrained floating-point expression `exp`, which is thus transformed into a constrained floating-point variable.

Class IlcFloatVarArray

Definition file: ilsolver/fltexp.h
Include file: <ilsolver/ilosolver.h>

The class `IlcFloatVarArray` is the class for an *array* of instances of `IlcFloatVar`. Three integers—`indexMin`, `indexMax`, and `indexStep`—play an important role in such an array of constrained floating-point variables. The index of those variables ranges from `indexMin`, inclusive, to `indexMax`, exclusive, in steps of `indexStep`. The index of the first variable in the array is `indexMin`; the second one is `indexMin+indexStep`, and so forth. The quantity indicated by `indexMax-indexMin` must be a multiple of `indexStep`.

Generic Constraints

The array makes it easier to implement *generic* constraints. In this context, a generic constraint is a constraint that applies to all of the variables in the array. Member functions of the array class are available to post such generic constraints. A generic constraint is then allocated and recorded only once for all the variables in the array. This fact represents a significant economy in memory, compared to allocating and recording one constraint per variable.

Interval Constraints

Arrays of constrained variables also allow you to define *interval constraints* which propagate in a global way when the domains of one or more constrained variables in the array are modified. Propagation is then performed through a goal. Member functions such as `whenValueInterval` or `whenRangeInterval` associate goals with propagation events for this purpose.

Backtracking and Reversibility

All the functions and member functions capable of modifying arrays of constrained floating-point variables are reversible. In particular, when modifiers and functions that post constraints are called, the state before their call will be saved by Solver.

Empty Handle or Null Array

It is possible to create a null array, or in other words, an empty handle. When you do so, only these operations are allowed on that null array:

- **copy**: You can assign the null array to a new array.
- **access its size**: The member function `getSize` for the null array returns 0 (zero).
- **create an iterator**: You can create an iterator to traverse the null array. The member function `ok` returns `IlcFalse` for a null array.

Attempts to access a null array in any other way will throw an exception (an instance of `IloSolver::SolverErrorException`).

See Also: IlcFloatVar, operator<<

Constructor Summary	
public	IlcFloatVarArray()
public	IlcFloatVarArray(IlcFloatVarArrayI * impl)
public	IlcFloatVarArray(IloSolver solver, IlcInt size)
public	IlcFloatVarArray(IloSolver solver, IlcInt size ILCPARAM, const IlcFloatVar v1, const IlcFloatVar v2)

public	IlcFloatVarArray(IloSolver m, IlcInt size, IlcFloat min, IlcFloat max)

Method Summary	
public IlcFloatVarArray	getCopy(IloSolver solver) const
public IlcFloatVarArrayI *	getImpl() const
public IlcInt	getIndexMax() const
public IlcInt	getIndexMin() const
public IlcInt	getIndexStep() const
public IlcInt	getIndexValue() const
public IlcFloat	getMaxMax() const
public IlcFloat	getMaxMax(IlcInt indexMin, IlcInt indexMax) const
public IlcFloat	getMaxMin() const
public IlcFloat	getMaxMin(IlcInt indexMin, IlcInt indexMax) const
public IlcFloat	getMinMax() const
public IlcFloat	getMinMax(IlcInt indexMin, IlcInt indexMax) const
public IlcFloat	getMinMin() const
public IlcFloat	getMinMin(IlcInt indexMin, IlcInt indexMax) const
public const char *	getName() const
public IlcInt	getRangeIndexMax() const
public IlcInt	getRangeIndexMin() const
public IlcInt	getSize() const
public IloSolver	getSolver() const
public IloSolverI *	getSolverI() const
public IlcInt	getValueIndexMax() const
public IlcInt	getValueIndexMin() const
public IlcFloatVar	getVariable(IlcInt index, IlcBool before=IlcFalse) const
public void	operator=(const IlcFloatVarArray & h)
public IlcFloatVar &	operator[](IlcInt index) const
public void	setName(const char * name) const
public void	whenRange(const IlcDemon demon)
public void	whenRangeInterval(const IlcDemon demon)
public void	whenValue(const IlcDemon demon)
public void	whenValueInterval(const IlcDemon demon)

Constructors

public **IlcFloatVarArray**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IlcFloatVarArray**(IlcFloatVarArrayI * impl)

This constructor creates a handle object from a pointer to an implementation object.

```
public IlcFloatVarArray(IloSolver solver, IlcInt size)
```

This constructor creates an uninitialized array of length `size`. The index range of the array is `[0 size)`. The argument `size` must be strictly greater than 0 (zero). Each element of the array must be assigned before the array can be used.

```
public IlcFloatVarArray(IloSolver solver, IlcInt size ILCPARAM, const IlcFloatVar
v1, const IlcFloatVar v2)
```

This constructor creates an array of length `size`. Its constrained variables are initialized with the list of variables provided as arguments to the constructor. The number of `IlcFloatVar` arguments must be equal to `size`. The argument `size` must be strictly greater than 0 (zero).

```
public IlcFloatVarArray(IloSolver m, IlcInt size, IlcFloat min, IlcFloat max)
```

This constructor creates an array of `size` constrained variables. The argument `size` must be strictly greater than 0 (zero). Each constrained variable has a domain containing all floating-point values between `min` and `max`.

Methods

```
public IlcFloatVarArray getCopy(IloSolver solver) const
```

This member function returns a copy of the invoking array of constrained variables and associates that copy with `solver`.

```
public IlcFloatVarArrayI * getImpl() const
```

This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

```
public IlcInt getIndexMax() const
```

This member function returns the maximal index of the invoking array of constrained variables.

```
public IlcInt getIndexMin() const
```

This member function returns the minimal index of the invoking array of constrained variables.

```
public IlcInt getIndexStep() const
```

This member function returns the index step of the invoking array of constrained variables. The meaning of this index step is that the indexed variable value may change only at indices equal to `(getIndexMin() + i * getIndexStep())`.

```
public IlcInt getIndexValue() const
```

When it is called during the execution of a constraint or goal associated with an array by the member functions `whenValue` or `whenDomain`, this member function returns the index in the invoking array of the constrained variable that triggered the propagation event. Calling this member function outside the execution of the goal will throw an exception (an instance of `IloSolver::SolverErrorException`) with the message "`unbound index`".

`public IlcFloat` **`getMaxMax`**`() const`

This member function returns the largest of the maximal values of the variables belonging to the invoking array of constrained variables.

`public IlcFloat` **`getMaxMax`**`(IlcInt indexMin, IlcInt indexMax) const`

This member function returns the largest of the maximal values of the variables belonging to the invoking array of constrained variables. The arguments `indexMin` and `indexMax` are provided, only those variables that correspond to indices in the range `[indexMin indexMax)` are considered. Solver will throw an exception (an instance of `IloSolver::SolverErrorException` with the message `"bad index interval"` if the given `indexMin` and `indexMax` are not valid indices for the invoking array of constrained variables or if `indexMin` is not strictly less than `indexMax`.

`public IlcFloat` **`getMaxMin`**`() const`

This member function returns the largest of the minimal values of the variables belonging to the invoking array of constrained variables.

`public IlcFloat` **`getMaxMin`**`(IlcInt indexMin, IlcInt indexMax) const`

This member function returns the largest of the minimal values of the variables belonging to the invoking array of constrained variables. The arguments `indexMin` and `indexMax` are provided, only those variables that correspond to indices in the range `[indexMin indexMax)` are considered. Solver will throw an exception (an instance of `IloSolver::SolverErrorException` with the message `"bad index interval"` if the given `indexMin` and `indexMax` are not valid indices for the invoking array of constrained variables or if `indexMin` is not strictly less than `indexMax`.

`public IlcFloat` **`getMinMax`**`() const`

This member function returns the smallest of the maximal values of the variables belonging to the invoking array of constrained variables.

`public IlcFloat` **`getMinMax`**`(IlcInt indexMin, IlcInt indexMax) const`

This member function returns the smallest of the maximal values of the variables belonging to the invoking array of constrained variables. The arguments `indexMin` and `indexMax` are provided, only those variables that correspond to indices in the range `[indexMin indexMax)` are considered. Solver will throw an exception (an instance of `IloSolver::SolverErrorException` with the message `"bad index interval"` if the given `indexMin` and `indexMax` are not valid indices for the invoking array of constrained variables or if `indexMin` is not strictly less than `indexMax`.

`public IlcFloat` **`getMinMin`**`() const`

This member function returns the smallest of the minimal values of the variables belonging to the invoking array of constrained variables.

`public IlcFloat **getMinMin**(IlcInt indexMin, IlcInt indexMax) const`

This member function returns the smallest of the minimal values of the variables belonging to the invoking array of constrained variables. The arguments `indexMin` and `indexMax` are provided, only those variables that correspond to indices in the range `[indexMin indexMax)` are considered. Solver will throw an exception (an instance of `IloSolver::SolverErrorException` with the message `"bad index interval"` if the given `indexMin` and `indexMax` are not valid indices for the invoking array of constrained variables or if `indexMin` is not strictly less than `indexMax`.

`public const char * **getName**() const`

This member function returns the name of the invoking object.

`public IlcInt **getRangeIndexMax**() const`

When it is called during the execution of a goal associated with an array by the member function `whenRangeInterval`, this member function returns the maximum of the range of the array `[indexMin indexMax)` over which some removal of values has occurred. The returned value of this member function is not meaningful outside the execution of a goal associated with the array by the member function `whenRangeInterval`.

`public IlcInt **getRangeIndexMin**() const`

When it is called during the execution of a goal associated with an array by the member function `whenRangeInterval`, this member function returns the minimum of the range of the array `[indexMin indexMax)` over which some removal of values has occurred. The returned value of this member function is not meaningful outside the execution of a goal associated with the array by the member function `whenRangeInterval`.

`public IlcInt **getSize**() const`

This member function returns the number of variables in the invoking array.

`public IloSolver **getSolver**() const`

This member function returns an instance of `IloSolver` associated with the invoking object.

`public IloSolverI * **getSolverI**() const`

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

`public IlcInt **getValueIndexMax**() const`

When it is called during the execution of a goal associated with an array by the member function `whenValueInterval`, this member function returns the maximum of the range of the array `[indexMin indexMax)` over which some binding has occurred. The returned value of this member function is not meaningful outside the execution of a goal associated with the array by the member function `whenValueInterval`.

`public IlcInt` **`getValueIndexMin`**`() const`

When it is called during the execution of a goal associated with an array by the member function `whenValueInterval`, this member function returns the minimum of the range of the array `[indexMin indexMax)` over which some binding has occurred. The returned value of this member function is not meaningful outside the execution of a goal associated with the array by the member function `whenValueInterval`.

`public IlcFloatVar` **`getVariable`**`(IlcInt index, IlcBool before=IlcFalse) const`

This member function returns the variable corresponding to the given `index` in the invoking array of constrained variables. However, if `before` is `IlcTrue`, then `getVariable` returns the variable *before* the variable at the given `index`. Solver will throw an exception (an instance of `IloSolver::SolverErrorException`) with the message "`bad index`" if the given `index` is not a valid one for the invoking array of constrained variables.

`public void` **`operator=`**`(const IlcFloatVarArray & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

`public IlcFloatVar &` **`operator[]`**`(IlcInt index) const`

This subscripting operator returns a reference to a constrained variable corresponding to the given `index` in the invoking array of constrained variables. Solver will throw an exception (an instance of `IloSolver::SolverErrorException`) with the message "`bad index`" if the given `index` is not a valid one for the invoking array of constrained variables.

`public void` **`setName`**`(const char * name) const`

This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

`public void` **`whenRange`**`(const IlcDemon demon)`

This member function associates `demon` with the range propagation event of every variable in the invoking array. Whenever any bound of any of the variables in the array is modified, the demon will be executed immediately.

When the demon is executed, the index of the constrained integer variable that has triggered the range event can be known by a call to the member function `getIndexValue`.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. Whenever any bound of any of the variables in the array is modified, the constraint will be propagated.

`public void` **`whenRangeInterval`**`(const IlcDemon demon)`

This member function associates `demon` with the range propagation event of every variable in the invoking array. It specifies that a given demon reacts globally to modifications of the boundaries of a collection of variables in the array. Whenever a range propagation event or a series of such events occurs, the demon will be executed immediately.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. Whenever a range propagation event or a series of such events occurs, the constraint will be propagated.

A call to the demon signifies that *some* modification of the boundaries occurred to variables over the index range `[indexMin indexMax)`. It does *not* mean that all the variables in the range had their boundaries modified.

`public void` **`whenValue`**`(const IlcDemon demon)`

This member function associates `demon` with the value propagation event of every variable in the invoking array. When one of the variables in the array receives a value, the demon is executed immediately.

When the demon is executed, the index of the bound constrained variable can be known by a call to the member function `getIndexValue`.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. When one of the variables in the array receives a value, the constraint will be propagated.

`public void` **`whenValueInterval`**`(const IlcDemon demon)`

This member function associates `demon` with the value propagation event of every variable in the invoking array. It specifies that a given demon reacts globally to the binding of a collection of variables in the array. When a value propagation event or a series of such events occurs, the demon will be executed immediately.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. When a value propagation event or a series of such events occurs, the constraint will be propagated.

A call to the demon signifies that *some* variable binding occurred over the index range `[indexMin indexMax)`. It does *not* mean that all the variables in the range have been bound.

Class IlcFloatVarDeltaIterator

Definition file: ilsolver/fltexp.h
Include file: <ilsolver/ilosolver.h>

An instance of the class `IlcFloatVarDeltaIterator` is an iterator that traverses the values belonging to the domain-delta of a constrained discrete floating-point variable (that is, an instance of `IlcFloatVar` created by the constructor `IlcFloatVar`).

A discrete floating-point variable has an enumerated domain. Consequently, its domain-delta is also enumerated and can thus be traversed by an iterator.

An iterator from this class will not traverse the domain-delta of a continuous floating-point variable (that is, one created by any of the other constructors of `IlcFloatVar`) because the domain and domain-delta of a continuous floating-point variable are represented by an interval (not by an enumerated set of values). Any attempt to traverse the domain-delta of a continuous floating-point variable will throw an exception (an instance of `IloSolver::SolverErrorException`).

For more information, see the concepts Propagation, Domain-Delta, and Iterator.

See Also: IlcFloatExp, IlcFloatVar

Constructor and Destructor Summary	
`public`	`IlcFloatVarDeltaIterator(const IlcFloatVar var)`

Method Summary	
`public IlcBool`	`ok() const`
`public IlcFloat`	`operator*() const`
`public IlcFloatVarDeltaIterator &`	`operator++()`

Constructors and Destructors

`public` **`IlcFloatVarDeltaIterator`**`(const IlcFloatVar var)`

This constructor creates an iterator associated with `var` to traverse the values belonging to the domain-delta of `var`.

Methods

`public IlcBool` **`ok`**`() const`

This member function returns `IlcTrue` if there is a current element and the iterator points to it. Otherwise, it returns `IlcFalse`.

To traverse the values belonging to the domain-delta of a constrained discrete floating-point variable, use the following code:

```
IlcFloat val;

for (IlcFloatVarDeltaIterator iter(var); iter.ok(); ++iter){
```

```
        val = *iter;
        // do something with val
}
```

```
public IlcFloat operator*() const
```

This operator returns the current element, the one to which the invoking iterator points.

```
public IlcFloatVarDeltaIterator & operator++()
```

This operator advances the iterator to point to the next value in the domain-delta of the constrained discrete floating-point variable.

Class IlcGoal

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>

Goals are the building blocks of search algorithms in Solver. Goals depend on two classes: `IlcGoal` and `IlcGoalI`. The class `IlcGoal` is the handle class. An instance of the class `IlcGoal` contains a data member (the handle pointer) that points to an instance of the class `IlcGoalI` (the implementation object) allocated on the Solver heap. If you define a new *class* of goals, you must define the implementation class together with the corresponding virtual member function, `execute`, and a member function that returns an instance of the handle class `IlcGoal`.

For more information, see the concept Goal.

See Also: IlcAnd, ILCGOAL0, IlcGoalI, IlcOr, IloGoalFail, operator<<

Constructor Summary	
`public`	`IlcGoal()`
`public`	`IlcGoal(IlcGoalI * impl)`
`public`	`IlcGoal(const IlcGoal & goal)`

Method Summary	
`public IlcGoalI *`	`getImpl() const`
`public const char *`	`getName() const`
`public IlcAny`	`getObject() const`
`public IloSolver`	`getSolver() const`
`public IloSolverI *`	`getSolverI() const`
`public void`	`operator=(const IlcGoal & h)`
`public void`	`setName(const char * name) const`
`public void`	`setObject(IlcAny object) const`

Constructors

`public` **`IlcGoal`**`()`

This constructor creates a goal which is empty, that is, one whose handle pointer is null. This object must then be assigned before it can be used, exactly as when you declare a pointer.

This constructor creates a handle object (an instance of the class `IlcGoal`) from a pointer to an implementation object (an instance of the implementation class `IlcGoalI`).

This member function returns a pointer to the implementation object of the invoking handle, a goal.

This member function returns the solver (a handle, an instance of `IloSolver`) associated with the invoking goal.

This assignment operator copies `goal` into the invoking goal by assigning an address to the handle pointer of the invoking object. That address is the location of the implementation object of the argument `goal`.

This constructor creates an empty handle. You must initialize it before you use it.

public **IlcGoal**(IlcGoalI * impl)

This constructor creates a handle object from a pointer to an implementation object.

public **IlcGoal**(const IlcGoal & goal)

This copy constructor creates a reference to a goal. That goal and `goal` both point to the same implementation object.

Methods

public IlcGoalI * **getImpl**() const

This constructor creates an object by copying another one. This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

public const char * **getName**() const

This member function returns the name of the invoking object.

public IlcAny **getObject**() const

This member function returns a pointer to the external object associated with the invoking object, if there is such an association. It returns 0 (zero) otherwise.

public IloSolver **getSolver**() const

This member function returns an instance of `IloSolver` associated with the invoking object.

public IloSolverI * **getSolverI**() const

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

public void **operator=**(const IlcGoal & h)

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

public void **setName**(const char * name) const

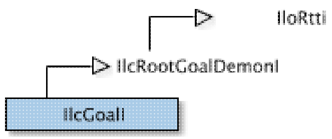
This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

public void **setObject**(IlcAny object) const

This member function establishes a link between the invoking object and an external object of which the invoking object might be a data member.

Class IlcGoalI

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>



Goals are the building blocks of search algorithms in Solver. Goals as they are represented in a Solver search (for example, inside a constraint or inside another goal) depend on two classes: `IlcGoal` and `IlcGoalI`. The class `IlcGoal` is the handle class. An instance of the class `IlcGoal` contains a data member (the handle pointer) that points to an instance of the class `IlcGoalI` (the implementation object) allocated on the Solver heap.

For goals to use in an IBM® ILOG® Concert Technology model, see `IloGoal`.

If you define a new *class* of goals for use during a Solver search, you must define the implementation class together with the corresponding virtual member function, `execute`, and a function that returns an instance of the handle class `IlcGoal`.

A goal can be defined in terms of other goals, called its *subgoals*. See the functions `IlcAnd` and `IlcOr` for examples. A subgoal is *not* executed immediately. In fact, it is added to the goal stack, and the `execute` member function of the current goal terminates before the subgoal is executed. When the execution of the goal itself is complete, the subgoal will be returned.

A goal can also be defined as a choice between other goals. This choice is implemented by the function `IlcOr`.

For more information, see the concept Goal.

See Also: ILCGOAL0, IlcGoal, operator<<

Constructor and Destructor Summary	
`public`	`IlcGoalI(IloSolver solver)`

Method Summary	
`public virtual IlcGoal`	`execute()`
`public void`	`fail(IlcAny label=0)`
`public IloSolver`	`getSolver() const`
`public IloSolverI *`	`getSolverI() const`
`public virtual IlcBool`	`isAConstraint() const`

Constructors and Destructors

`public` **`IlcGoalI`**`(IloSolver solver)`

This constructor creates a goal implementation. This constructor should not be called directly because this is an abstract class. This constructor is called automatically in the constructors of its subclasses.

Methods

```
public virtual IlcGoal execute()
```

This member function must be redefined when you derive a new subclass of `IlcGoalI`. This member function is called when the invoking goal is popped from the goal stack and executed. This member function should return 0 if the invoking goal has no subgoals. Otherwise it should return the subgoal of the goal.

Example

Here's how to define a class of goals without subgoals. This goal merely prints the integer passed as its argument.

```
class PrintXI :public IlcGoalI {
    IlcInt x;
  public:
    PrintXI(IloSolver s, IlcInt xx): IlcGoal(s), x(xx){}
    ~PrintXI(){}
    IlcGoal execute() {
        IloSolver s = getSolver();
        s.out() << "PrintX: a goal with one data member" << endl;
        s.out() << x << endl;
        return 0;
    }
};

IlcGoal PrintX(IloSolver s, IlcInt x) {
    return new (s.getHeap()) PrintXI(s, x);
}
```

The macro `ILCGOAL` makes it easier to define goals.

Here's an example of a goal with one subgoal:

```
ILCGOAL0(Print()){
    IloSolver s = getSolver();
    s.out() << "before one subgoal" << endl;
    return PrintX(s,2);
}
```

A goal can also be defined as a choice between other goals. This choice is implemented by the function `IlcOr`. For example, the following goal has three choices:

```
ILCGOAL0(PrintOne) {
    IloSolver s = getSolver();
    s.out() << "print one" << endl;
    return IlcOr(PrintX(s,1), PrintX(s,2), PrintX(s,3)));
}
```

```
public void fail(IlcAny label=0)
```

This member function causes the invoking goal to fail. The optional argument `label` makes the invoking goal fail at the choice point named `label`.

```
public IloSolver getSolver() const
```

This member function returns the solver (a handle) of the invoking goal implementation.

```
public IloSolverI * getSolverI() const
```

This member function returns a pointer to the implementation object of the solver where the invoking goal was extracted.

```
public virtual IlcBool isAConstraint() const
```

This member function lets you know whether the active demon is a constraint (in that case, it returns `IlcTrue` or a goal (in that case, it returns `IlcFalse`).

Class IlcIndex

Definition file: ilsolver/index.h
Include file: <ilsolver/ilosolver.h>

To create a generic constraint, you need generic variables. In order to create generic variables for an array of constrained expressions, Solver provides this class `IlcIndex`.

A *generic constraint* is a constraint shared by an array of variables. For example, `IlcAllDiff` is a generic constraint that insures that all the elements of a given array are different from one another. Solver provides generic constraints to save memory since, if you use them, you can avoid allocating one object per variable.

You create a generic constraint simply by stating the constraint over *generic variables*. Each generic variable stands for all the elements of an *array* of constrained variables.

In that sense, generic variables are only syntactic objects provided by Solver to support generic constraints, and they can be used only for creating generic constraints. To create a generic variable, you use the operator `[]`. The argument passed to that operator is known as the *index* for that generic variable; we say that the generic variable *stems from* that index.

An index is simply a syntactical means for creating generic variables for an array of constrained variables. The index has no value, and no value can be assigned to it. Any attempt to access the value of an index will throw an exception (an instance of `IloSolver::SolverErrorException`).

See Also: IlcAnyVarArray, IlcCard, IlcIntVarArray, IlcSetOf

Constructor Summary	
public	IlcIndex(IlcIndexI * impl)
public	IlcIndex(IloSolver solver)

Method Summary	
public IlcIndexI *	getImpl() const

Constructors

public **IlcIndex**(IlcIndexI * impl)

This constructor creates a handle object (an instance of the class `IlcIndex`) from a pointer to an object (an instance of the implementation class `IlcIndexI`).

public **IlcIndex**(IloSolver solver)

This constructor creates an index which will be managed by `solver`.

Methods

public IlcIndexI * **getImpl**() const

This member function returns a pointer to the implementation object of the invoking handle, an index.

Class IlcIntArray

Definition file: ilsolver/intexp.h
Include file: <ilsolver/ilosolver.h>

`IlcIntArray` is the array class for the basic integer class. It is a handle class.

For each basic type, Solver defines a corresponding array class. This array class is a handle class. In other words, an object of this class contains a pointer to another object allocated on the Solver heap associated with a solver (an instance of `IloSolver`). Exploiting handles in this way greatly simplifies the programming interface since the handle can then be an automatic object: as a developer using handles, you do not have to worry about memory allocation.

Empty Handle or Null Array

It is possible to create a null array, or in other words, an empty handle. When you do so, only these operations are allowed on that null array:

- **copy**: You can assign the null array to a new array.
- **access its size**: The member function `getSize` for the null array returns 0 (zero).
- **create an iterator**: You can create an iterator to traverse the null array. The member function `ok` returns `IlcFalse` for a null array.

Attempts to access a null array in any other way will throw an exception (an instance of `IloSolver::SolverErrorException`).

See Also: IlcConstIntArray, IlcIntExp, IlcIntSet, operator<<

Constructor Summary	
public	IlcIntArray()
public	IlcIntArray(IlcInt * impl)
public	IlcIntArray(IloSolver solver, IlcInt size, IlcInt * values)
public	IlcIntArray(IloSolver solver, IlcInt size, IlcInt prototype=0)
public	IlcIntArray(IloSolver solver, IlcInt size, IlcInt exp0, IlcInt exp...)

Method Summary	
public IlcInt *	getImpl() const
public IlcInt	getSize() const
public IloSolver	getSolver() const
public void	operator=(const IlcIntArray & h)
public IlcIntExp	operator[](const IlcIntExp rank) const
public IlcInt &	operator[](IlcInt i) const

Constructors

public **IlcIntArray**()

This constructor creates an empty handle. You must initialize it before you use it.

```
public IlcIntArray(IlcInt * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IlcIntArray(IloSolver solver, IlcInt size, IlcInt * values)
```

This constructor creates an array of integers containing the values in the array `values`. The argument `size` must be the length of the array `values`; it must also be strictly greater than 0 (zero). Solver does not keep a pointer to the array `values`.

Here is one way to create an array containing the integers 1, 3, 2.

```
IlcInt values [3];
values[0] = 1;
values[1] = 3;
values[2] = 2;
IlcIntArray array1 (s, 3, values);
```

```
public IlcIntArray(IloSolver solver, IlcInt size, IlcInt prototype=0)
```

This constructor creates an array of `size` elements. The argument `size` must be strictly greater than 0 (zero). The elements of this array are initialized to the value of `prototype`.

```
public IlcIntArray(IloSolver solver, IlcInt size, IlcInt exp0, IlcInt exp...)
```

This constructor accepts a variable number of arguments. Its first argument, `size`, indicates the length of the array that this constructor will create; `size` must be the number of arguments minus one; it must also be strictly greater than 0 (zero). The constructor creates an array of the values indicated by the other arguments.

Here is another way to create an array containing the integers 1,3,2.

```
IlcIntArray array2 (s, 3, 1, 3, 2);
```

Methods

```
public IlcInt * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IlcInt getSize() const
```

This member function returns the number of elements in the array.

```
public IloSolver getSolver() const
```

This member function returns an instance of `IloSolver` associated with the invoking object.

`public void operator=(const IlcIntArray & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

`public IlcIntExp operator[](const IlcIntExp rank) const`

This subscripting operator returns a constrained integer expression. For clarity, let's call `A` the invoking array. When `rank` is bound to the value `i`, the value of the expression is `A[i]`. More generally, the domain of the expression is the set of values `A[i]` where the `i` are in the domain of `rank`.

`public IlcInt & operator[](IlcInt i) const`

This operator returns a reference to the element at rank `i`. This operator can be used for accessing (that is, simply reading) the element or for modifying (that is, writing) it.

Here is still another way to create an array containing the integers 1, 3, 2.

```
IlcIntArray array5 (s, 3);
array5[0] = 1;
array5[1] = 3;
array5[2] = 2;
```

Class IlcIntDeltaPossibleIterator

Definition file: ilsolver/ilcset.h
Include file: <ilsolver/ilosolver.h>

An instance of the class `IlcIntDeltaPossibleIterator` is an iterator that traverses the elements of the possible delta set of an instance of `IlcIntSetVar` (a constrained set variable). The order in which the iterator traverses the possible delta set is not predictable.

For more information, see the concepts Propagation, Domain-Delta, and Iterator.

See Also: IlcIntDeltaRequiredIterator, IlcIntSetVar

Constructor Summary	
public	IlcIntDeltaPossibleIterator(IlcIntSetVar var)

Method Summary	
public IlcBool	ok() const
public IlcInt	operator*() const
public IlcIntDeltaPossibleIterator &	operator++()

Constructors

public **IlcIntDeltaPossibleIterator**(IlcIntSetVar var)

This constructor creates an iterator associated with `var` to traverse the values belonging to its possible delta set.

Methods

public IlcBool **ok**() const

This member function returns `IlcTrue` if there is a current element and the iterator points to it. Otherwise, it returns `IlcFalse`.

public IlcInt **operator***() const

This operator returns the current element, the one to which the invoking iterator points.

public IlcIntDeltaPossibleIterator & **operator++**()

This operator advances the iterator to point to the next value in the possible delta set.

Class IlcIntDeltaRequiredIterator

Definition file: ilsolver/ilcset.h
Include file: <ilsolver/ilosolver.h>

An instance of the class `IlcIntDeltaRequiredIterator` is an iterator that traverses the elements of the required delta set of an instance of `IlcIntSetVar` (a constrained set variable). The order in which the iterator traverses the required delta set is not predictable.

For more information, see the concepts Propagation, Domain-Delta, and Iterator.

See Also: IlcIntDeltaPossibleIterator, IlcIntSetVar

Constructor Summary	
public	IlcIntDeltaRequiredIterator(IlcIntSetVar var)

Method Summary	
public IlcBool	ok() const
public IlcInt	operator*() const
public IlcIntDeltaRequiredIterator &	operator++()

Constructors

public **IlcIntDeltaRequiredIterator**(IlcIntSetVar var)

This constructor creates an iterator associated with `var` to traverse the values belonging to its required delta set.

Methods

public IlcBool **ok**() const

This member function returns `IlcTrue` if there is a current element and the iterator points to it. Otherwise, it returns `IlcFalse`.

public IlcInt **operator***() const

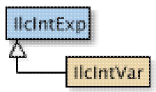
This operator returns the current element, the one to which the invoking iterator points.

public IlcIntDeltaRequiredIterator & **operator++**()

This operator advances the iterator to point to the next value in the required delta set.

Class IlcIntExp

Definition file: ilsolver/intexp.h
Include file: <ilsolver/ilosolver.h>



In a typical application exploiting Solver, the unknowns of the problem will be expressed as constrained variables. The most commonly used class of constrained variables is the class of constrained *integer* variables. `IlcIntExp`, the class of constrained integer expressions, is the root class of a group of classes for expressing constraints on integer variables.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Domain

The domain of a constrained integer expression is *computed* from the domains of its subexpressions. For example, the domain of the expression `x+y` contains the range `[x.getMin()+y.getMin(), x.getMax()+y.getMax()]`.

A constrained integer variable is a constrained expression that *stores* its domain instead of computing it from its subexpressions. The domain of a constrained integer variable contains values of type `IlcInt`. This domain is represented by an interval when the values are consecutive or by an enumeration of integers otherwise.

Constrained integer variables can be combined with arithmetic operators to yield constrained integer expressions. Each constrained integer expression has a minimum and a maximum. We say that the expression is *bound* if its minimum equals its maximum.

The domain of a constrained integer expression can be reduced to the point of being empty. In such a case, failure occurs since no solution is then possible.

Expression versus Variable

`IlcIntVar` is a subclass deriving from `IlcIntExp`. Another way of saying that idea is that a constrained variable is a constrained expression that happens to store its domain. You can convert a constrained integer expression (which computes its domain) into a constrained integer variable (which stores its domain) by either of two means: by the casting constructor of `IlcIntVar` or by the assignment operator of `IlcIntVar`. For more information, see *Chapter 3, "Constrained Integer Variables,"* in the *IBM ILOG Solver User's Manual*.

Overflow and Underflow

Previous versions of Solver did not check for integer overflow nor underflow, and we formerly recommended that users with concerns about overflow and underflow should use floating-point variables. However, as of version 5, Solver manages integer overflow and underflow in these ways:

- The arithmetic operators `+`, `*`, `-`, `/` do not cause overflow in Solver.
- The member function `IlcIntExp::getSize` returns `IlcIntMax` whenever `max - min` is greater than `IlcIntMax`.
- If an integer expression overflows negatively (that is, if a bound is less than `IlcIntMin`), then Solver replaces that bound by `IlcIntMin`.
- If an integer expression overflows positively (that is, if a bound is greater than `IlcIntMax`) then Solver replaces that bound by `IlcIntMax`.
- The value `IlcIntMin - 1` (sometimes known as the Joker) is treated correctly as long as you do not use it in expressions in `+`, `*`, `-`, `/`.
- Solver evaluates the expression `0/0` as the interval `[IlcIntMin..IlcIntMax]`.

Backtracking and Reversibility

All the member functions and operators defined for this class and capable of modifying constrained variables are *reversible*. In particular, the changes made by constraint-posting functions are made with reversible assignments. Thus, the value, the domain, and the constraints posted on any constrained variable are restored when Solver backtracks.

Modifiers

For modifiers of `IlcIntExp`, see the concept Propagation Events.

For more information, see the concept Propagation.

A modifier is a member function that reduces the domain of a constrained integer expression, if it can. The modifier is not stored, in contrast to a constraint. If the constrained integer expression is a constrained integer variable, the modifications due to the modifier call are stored in its domain. Otherwise, the effect of the modifier is propagated to the subexpressions of the constrained integer expression. If the domain becomes empty, a failure is triggered by a call to the member function `IloSolver::fail`. Modifiers are usually used to define new *classes* of constraints.

See Also: IlcAbs, IlcIntExpIterator, IlcIntSet, IlcIntVar, IlcIntVarArray, IlcMax, IlcMin, IlcScalProd, IlcSquare, IlcSum, operator+, operator/, operator*, operator-, operator-, operator<<

Constructor Summary	
public	IlcIntExp()
public	IlcIntExp(IlcIntExpI * impl)
public	IlcIntExp(IlcConstraint bexp)
public	IlcIntExp(IlcBoolVar bexp)

Method Summary	
public IlcIntExp	getCopy(IloSolver solver) const
public IlcIntExpI *	getImpl() const
public IlcInt	getMax() const
public IlcInt	getMin() const
public const char *	getName() const
public IlcInt	getNextHigher(IlcInt threshold) const
public IlcInt	getNextLower(IlcInt threshold) const
public IlcAny	getObject() const
public IlcInt	getSize() const
public IloSolver	getSolver() const
public IloSolverI *	getSolverI() const
public IlcInt	getValue() const
public IlcBool	isBound() const
public IlcBool	isInDomain(IlcInt value) const
public void	operator=(const IlcIntExp & h)
public void	removeDomain(IlcIntSet set)
public void	removeDomain(IlcIntArray array)
public void	removeRange(IlcInt min, IlcInt max) const
public void	removeValue(IlcInt value) const
public void	setDomain(IlcIntArray array)

public void	setDomain(IlcIntSet set)
public void	setDomain(IlcIntExp var)
public void	setMax(IlcInt max) const
public void	setMin(IlcInt min) const
public void	setName(const char * name) const
public void	setObject(IlcAny object) const
public void	setRange(IlcInt min, IlcInt max) const
public void	setValue(IlcInt value) const
public void	whenDomain(const IlcDemon demon) const
public void	whenRange(const IlcDemon demon) const
public void	whenValue(const IlcDemon demon) const

Constructors

public **IlcIntExp**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IlcIntExp**(IlcIntExpI * impl)

This constructor creates a handle object from a pointer to an implementation object.

public **IlcIntExp**(IlcConstraint bexp)

A constrained Boolean expression in Solver can be seen as a 0-1 (that is, binary) constrained integer expression where `IlcFalse` corresponds to 0, and `IlcTrue` corresponds to 1. This constructor creates a constrained integer expression which is equal to the truth value of the argument `bexp`. In other words, you can use this constructor to cast a constraint to a constrained integer expression.

Such a constrained integer expression can then be used like any other constrained integer expression. It is thus possible to express sums of constraints. For example, the following code expresses the idea that three variables cannot all be equal.

```
m.add((x != y) + (y != z) + (z != x) >= 2);
```

public **IlcIntExp**(IlcBoolVar bexp)

A constrained Boolean variable in Solver can be seen as a 0-1 (that is, binary) constrained integer expression where `IlcFalse` corresponds to 0, and `IlcTrue` corresponds to 1. This constructor creates a constrained integer expression which is equal to the truth value of the argument `bexp`. In other words, -* you can use this constructor to cast a Boolean variable to a 8 constrained integer expression.

Methods

public IlcIntExp **getCopy**(IloSolver solver) const

This member function returns a copy of the invoking constrained integer expression and associates that copy with `solver`.

public IlcIntExpI * **getImpl**() const

This constructor creates an object by copying another one. This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

`public IlcInt getMax() const`

This member function returns the maximum of the domain of the invoking object.

`public IlcInt getMin() const`

This member function returns the minimum of the domain of the invoking object.

`public const char * getName() const`

This member function returns the name of the invoking object.

`public IlcInt getNextHigher(IlcInt threshold) const`

If `threshold` is greater than or equal to the maximum of the domain of the invoking constrained integer expression, then this member function returns `threshold`. Otherwise, it returns the first element that is strictly greater than `threshold` in the domain of the invoking constrained integer expression.

`public IlcInt getNextLower(IlcInt threshold) const`

If `threshold` is less than or equal to the minimum of the domain of the invoking constrained integer expression, then this member function returns `threshold`. Otherwise, it returns the first element that is strictly less than `threshold` in the domain of the invoking constrained integer expression.

`public IlcAny getObject() const`

This member function returns a pointer to the external object associated with the invoking object, if there is such an association. It returns 0 (zero) otherwise.

`public IlcInt getSize() const`

This member function returns the number of elements in the domain of the invoking expression. In particular, it returns 1 if the invoking constrained integer expression is bound, and it returns `IlcIntMax` whenever `max - min` is greater than `IlcIntMax`.

`public IloSolver getSolver() const`

This member function returns an instance of `IloSolver` associated with the invoking object.

`public IloSolverI * getSolverI() const`

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

`public IlcInt getValue() const`

This member function returns the value of the invoking constrained integer expression if that object is bound; otherwise, Solver will throw an exception (an instance of `IloSolver::SolverErrorException`). To avoid errors with `getValue`, you can test expressions by means of `isBound`.

`public IlcBool isBound() const`

This member function returns `IlcTrue` if the invoking constrained integer expression is bound, that is, if its minimum equals its maximum. Otherwise, the member function returns `IlcFalse`.

`public IlcBool isInDomain(IlcInt value) const`

This member function returns `IlcTrue` if `value` is in the domain of the invoking constrained integer expression. Otherwise, the member function returns `IlcFalse`.

`public void operator=(const IlcIntExp & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

`public void removeDomain(IlcIntSet set)`

This member function removes all the elements of the set indicated by `set` from the domain of the invoking constrained expression. If the domain thus becomes empty, then the member function `IloSolver::fail` is called. Otherwise, if the domain is modified, the corresponding propagation events are generated. The effects of this member function are reversible.

`public void removeDomain(IlcIntArray array)`

This member function removes all the elements of the array indicated by `array` from the domain of the invoking constrained expression. If the domain thus becomes empty, then the member function `IloSolver::fail` is called. Otherwise, if the domain is modified, the corresponding propagation events are generated. The effects of this member function are reversible.

`public void removeRange(IlcInt min, IlcInt max) const`

This member function removes all the elements that are both greater than or equal to `min` and less than or equal to `max` from the domain of the invoking constrained expression. If the domain thus becomes empty, then the member function `IloSolver::fail` is called. Otherwise, if the domain is modified, the domain propagation event is generated. If `min` or `max` was one of the bounds of the domain, then the range propagation event is generated, too. Moreover, if the invoking constrained integer expression becomes bound, then the value propagation event is also generated. The effects of this member function are reversible.

`public void removeValue(IlcInt value) const`

This member function removes `value` from the domain of the invoking constrained expression. If the domain thus becomes empty, then the member function `IloSolver::fail` is called. Otherwise, if the domain is modified, the domain propagation event is generated. If `value` was one of the bounds of the domain, then the range propagation event is generated, too. Moreover, if the invoking constrained integer expression becomes bound, then the value propagation event is also generated. The effects of this member function are reversible.

`public void` **`setDomain`**`(IlcIntArray array)`

This member function removes all the elements that are not in the array indicated by `array` from the domain of the invoking constrained expression. If the domain thus becomes empty, then the member function `IloSolver::fail` is called. Otherwise, if the domain is modified, the corresponding propagation events are generated. The effects of this member function are reversible.

`public void` **`setDomain`**`(IlcIntSet set)`

This member function removes all the elements that are not in the set indicated by `set` from the domain of the invoking constrained expression. If the domain thus becomes empty, then the member function `IloSolver::fail` is called. Otherwise, if the domain is modified, the corresponding propagation events are generated. The effects of this member function are reversible.

`public void` **`setDomain`**`(IlcIntExp var)`

This member function removes all the elements that are not in domain of `var` from the domain of the invoking constrained expression. If the domain thus becomes empty, then the member function `IloSolver::fail` is called. Otherwise, if the domain is modified, the corresponding propagation events are generated. The effects of this member function are reversible.

`public void` **`setMax`**`(IlcInt max) const`

This member function removes all the elements that are greater than `max` from the domain of the invoking constrained expression. If the domain thus becomes empty, then the member function `IloSolver::fail` is called. Otherwise, if the domain is modified, the range and domain propagation events are generated. Moreover, if the invoking constrained integer expression becomes bound, then the value propagation event is also generated. The effects of this member function are reversible.

`public void` **`setMin`**`(IlcInt min) const`

This member function removes all the elements that are less than `min` from the domain of the invoking constrained expression. If the domain thus becomes empty, then the member function `IloSolver::fail` is called. Otherwise, if the domain is modified, the range and domain propagation events are generated. Moreover, if the invoking constrained integer expression becomes bound, then the value propagation event is also generated. The effects of this member function are reversible.

`public void` **`setName`**`(const char * name) const`

This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

`public void` **`setObject`**`(IlcAny object) const`

This member function establishes a link between the invoking object and an external object of which the invoking object might be a data member.

`public void setRange(IlcInt min, IlcInt max) const`

This member function removes all the elements that are either less than `min` or greater than `max` from the domain of the invoking constrained expression. If the domain thus becomes empty, then the member function `IloSolver::fail` is called. Otherwise, if the domain is modified, the propagation events `range` and `domain` are generated. Moreover, if the invoking constrained integer expression becomes bound, then the value propagation event is also generated. The effects of this member function are reversible.

`public void setValue(IlcInt value) const`

This member function removes all the elements that are different from `value` from the domain of the invoking constrained integer expression. This has two possible outcomes:

- If `value` was not in the domain of the invoking constrained integer expression, the domain becomes empty, and the member function `IloSolver::fail` is called.
- If `value` was in the domain, then `value` becomes the value of the expression, and the value, range, and domain propagation events are generated.

The effects of this member function are reversible.

`public void whenDomain(const IlcDemon demon) const`

This member function associates `demon` with the domain event of the invoking constrained expression. Whenever the domain of the invoking constrained expression changes, the demon will be executed immediately.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. Whenever the domain of the invoking constrained expression changes, the constraint will be propagated.

`public void whenRange(const IlcDemon demon) const`

This member function associates `demon` with the range event of the invoking constrained expression. Whenever one of the bounds of the domain of the invoking constrained expression changes, the demon will be executed immediately.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. Whenever one of the bounds of the domain of the invoking constrained expression changes, the constraint will be propagated.

`public void whenValue(const IlcDemon demon) const`

This member function associates `demon` with the value event of the invoking constrained expression. Whenever the invoking constrained expression becomes bound, the demon will be executed immediately.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. Whenever the invoking constrained expression becomes bound, the constraint will be propagated.

Class IlcIntExpIterator

Definition file: ilsolver/intexp.h
Include file: <ilsolver/ilosolver.h>

An instance of the class `IlcIntExpIterator` is an iterator that traverses the values belonging to the domain of a constrained integer expression (an instance of `IlcIntExp` or `IlcIntVar`).

For more information, see the concept Iterator.

See Also: IlcIntExp, IlcIntVar

Constructor Summary	
public	IlcIntExpIterator(IlcIntExp exp)

Method Summary	
public IlcBool	ok() const
public IlcInt	operator*() const
public IlcIntExpIterator &	operator++()

Constructors

public **IlcIntExpIterator**(IlcIntExp exp)

This constructor creates an iterator associated with `exp` to traverse the values belonging to the domain of `exp`.

Methods

public IlcBool **ok**() const

This member function returns `IlcTrue` if there is a current element and the iterator points to it. Otherwise, it returns `IlcFalse`.

To traverse the values belonging to the domain of a constrained integer expression, use the following code:

```
IlcInt val;
for (IlcIntExpIterator iter(exp); iter.ok(); ++iter){
        val = *iter;
        // do something with val
}
```

public IlcInt **operator***() const

This operator returns the current element, the one to which the invoking iterator points.

public IlcIntExpIterator & **operator++**()

This operator advances the iterator to point to the next value in the domain of the constrained integer expression.

Class IlcIntPredicate

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This class makes it possible for you to define integer predicates. An integer predicate is an object with a method (`IlcIntPredicate::isTrue`) that checks whether or not a property is satisfied by an ordered set of integers. The ordered set of integers is conventionally represented in Solver by an instance of `IlcIntArray`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Defining a New Class of Integer Predicates

Integer predicates, like other Solver objects, depend on two classes: a handle class, `IlcIntPredicate`, and an implementation class, `IlcIntPredicateI`, where an object of the handle class contains a data member (the handle pointer) that points to an object (its implementation object), an instance of `IlcIntPredicateI` allocated on the Solver heap. As a Solver user, you will be working primarily with handles.

If you define a new class of integer predicates yourself, you must define its implementation class together with the corresponding virtual member function `IlcIntPredicateI::isTrue`, as well as a member function that returns an instance of the handle class `IlcIntPredicate`.

Arity

As a developer, you can use predicates in Solver applications to define your own constraints that have not already been predefined in Solver. In that case, the *arity* of the predicate (that is, the number of constrained variables involved in the predicate, and thus the size of the array that the member function `IlcIntPredicate::isTrue` must check) must be less than or equal to three.

See Also: ILCANYPREDICATE0, IlcIntArray, ILCINTPREDICATE0, IlcIntPredicateI, IlcTableConstraint

Constructor Summary	
`public`	`IlcIntPredicate()`
`public`	`IlcIntPredicate(IlcIntPredicateI * impl)`

Method Summary	
`public IlcIntPredicateI *`	`getImpl() const`
`public IlcBool`	`isTrue(IlcIntArray val)`
`public void`	`operator=(const IlcIntPredicate & h)`

Constructors

`public` **`IlcIntPredicate`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcIntPredicate`**`(IlcIntPredicateI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

Methods

`public IlcIntPredicateI * getImpl() const`

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

`public IlcBool isTrue(IlcIntArray val)`

This member function calls the member function `isTrue` of the implementation class `IlcIntPredicateI`.

`public void operator=(const IlcIntPredicate & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IlcIntPredicateI

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

`IlcIntPredicateI` is the implementation class of `IlcIntPredicate`, which makes it possible for you to define integer predicates in Solver. An *integer predicate* is an object with a method (`IlcIntPredicateI::isTrue`) that checks whether or not a property is satisfied by an ordered set of integers. Conventionally in Solver, the ordered set of integers is represented by an instance of `IlcIntArray`.

Defining Your Own Class of Integer Predicates

Integer predicates, like other Solver objects, depend on two classes: a handle class, `IlcIntPredicate`, and an implementation class, `IlcIntPredicateI`, where an object of the handle class contains a data member (the handle pointer) that points to an object (its implementation object), an instance of `IlcIntPredicateI` allocated on the Solver heap. As a Solver user, you will be working primarily with handles.

If you define a new class of integer predicates yourself, you must define its implementation class together with the corresponding virtual member function `IlcIntPredicateI::isTrue`, as well as a member function that returns an instance of the handle class `IlcIntPredicate`.

Arity

As a developer, you can use predicates in Solver applications to define your own constraints that have not already been predefined in Solver. In that case, the *arity* of the predicate (that is, the number of constrained variables involved in the predicate, and thus the size of the array that the member function `IlcIntPredicateI::isTrue` must check) must be less than or equal to three.

See Also: IlcIntArray, IlcIntPredicate, IlcTableConstraint

Constructor and Destructor Summary	
`public`	`IlcIntPredicateI()`
`public`	`~IlcIntPredicateI()`

Method Summary	
`public virtual IlcBool`	`isTrue(IlcIntArray val)`

Constructors and Destructors

`public` **`IlcIntPredicateI`**`()`

This constructor creates an implementation object of an integer predicate. This constructor should not be called directly because this is an abstract class. This constructor is called automatically in the constructors of its subclasses.

`public` **`~IlcIntPredicateI`**`()`

As this class is to be subclassed, a virtual destructor is provided.

Methods

```
public virtual IlcBool isTrue(IlcIntArray val)
```

This member function must be redefined when you derive a new subclass of `IlcIntPredicateI`. This member function must return `IlcTrue` if the invoking predicate is satisfied by the elements contained in the array `val`. Otherwise, it must return `IlcFalse`.

Class IlcIntSelect

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

Solver lets you control the order in which the values in the domain of a constrained variable are tried during the search for a solution.

This class is the handle class of the object that chooses the value to try when the constrained variable under consideration is a constrained integer variable (that is, an instance of `IlcIntVar`).

An object of this handle class uses the virtual member function `IlcIntSelectI::select` from its implementation class to choose a value in the domain of the constrained integer variable under consideration during the search for a solution.

See Also: IlcEvalInt, IlcIntSelectI, IlcIntSelectEvalI, IloIntValueSelector, IloIntValueSelectorI

Constructor Summary	
`public`	`IlcIntSelect()`
`public`	`IlcIntSelect(IlcIntSelectI * impl)`
`public`	`IlcIntSelect(const IlcIntSelect & selector)`
`public`	`IlcIntSelect(IloSolver s, IlcEvalInt function)`

Method Summary	
`public IlcIntSelectI *`	`getImpl() const`
`public void`	`operator=(const IlcIntSelect & h)`

Constructors

`public` **`IlcIntSelect`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcIntSelect`**`(IlcIntSelectI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public` **`IlcIntSelect`**`(const IlcIntSelect & selector)`

This copy constructor accepts a reference to an implementation object and creates the corresponding handle object.

`public` **`IlcIntSelect`**`(IloSolver s, IlcEvalInt function)`

This constructor creates a new integer selector from an evaluation function. The implementation object of the newly created handle is an instance of the class `IlcIntSelectEvalI` constructed with the evaluation function indicated by the argument `function`.

Methods

```
public IlcIntSelectI * getImpl() const
```

This constructor creates an object by copying another one.

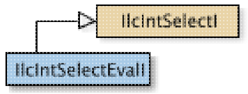
This member function returns a pointer to the implementation object of the invoking handle.

```
public void operator=(const IlcIntSelect & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IlcIntSelectEvalI

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>



Solver lets you control the order in which the values in the domain of a constrained variable are tried during the search for a solution.

This class is an implementation class. It is the predefined subclass of `IlcIntSelectI` that you use for the purpose of defining a new choice criterion expressed by an evaluation function on the domain of a constrained integer variable under consideration during the solution search. That evaluation function is of type `IlcEvalInt` to make the choice on the domain of a constrained integer variable.

See Also: IlcEvalInt, IlcIntSelect, IlcIntSelectI

Constructor Summary	
`public`	`IlcIntSelectEvalI(IlcEvalInt function)`

Method Summary	
`public virtual IlcInt`	`select(IlcIntVar var)`

Inherited Methods from `IlcIntSelectI`
`select`

Constructors

`public` **`IlcIntSelectEvalI`**`(IlcEvalInt function)`

This constructor creates an implementation object accompanied by an evaluation function. Objects of this class use that evaluation function to choose a value from the domain of a constrained integer variable during the search for a solution.

Methods

`public virtual IlcInt` **`select`**`(IlcIntVar var)`

This virtual member function returns one of the values of the domain of the constrained integer variable `var`. In order to do this, it calls the evaluation function for each value in the domain of the constrained integer variable `var`. For each of these values, the evaluation function is called with the value and the variable as arguments. Then the member function `select` returns the value for which the evaluation function returned the smallest integer.

Class IlcIntSelectI

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

Solver lets you control the order in which the values in the domain of a constrained variable are tried during the search for a solution.

This class is the implementation class for `IlcIntSelect`, the class of the object that chooses the value to try when the constrained variable under consideration is a constrained integer variable (that is, an instance of `IlcIntVar`).

The virtual member function in `IlcIntSelectI` chooses a value in the domain of the constrained integer variable under consideration.

To define new selection criteria, you can define a subclass of `IlcIntSelectI`. If those criteria can be expressed by an evaluation function, then you can use the predefined subclass `IlcIntSelectEvalI` for that purpose.

See Also: IlcEvalInt, IlcIntSelect, IlcIntSelectEvalI, IloIntValueSelector, IloIntValueSelectorI

Constructor and Destructor Summary	
public	IlcIntSelectI()
public	~IlcIntSelectI()

Method Summary	
public virtual IlcInt	select(IlcIntVar var)

Constructors and Destructors

public **IlcIntSelectI**()

This constructor creates an implementation object.

public **~IlcIntSelectI**()

As this class is to be subclassed, a virtual destructor is provided.

Methods

public virtual IlcInt **select**(IlcIntVar var)

This virtual member function returns one of the values of the domain of the constrained integer variable `var`. Its default implementation for the class `IlcIntSelectI` returns the minimum of the domain.

Class IlcIntSet

Definition file: ilsolver/intexp.h
Include file: <ilsolver/ilosolver.h>

Finite sets of integers are instances of the handle class `IlcIntSet`. These sets are used by Solver to represent the domains of enumerated constrained variables and to represent the values of constrained set variables. Solver provides an efficient, optimized implementation of finite sets as bit vectors. These finite sets are known formally as instances of the handle classes `IlcAnySet` or `IlcIntSet`, depending on whether their elements are pointers or integers.

The elements of finite sets of type `IlcIntSet` are integers of type `IlcInt`. The implementation class for finite sets of integers is the undocumented class `IlcIntSetI`.

To traverse an existing finite set, either exhaustively or in search of an element, Solver provides iterators (such as instances of `IlcIntSetIterator`). An iterator is an object constructed from a data structure and contains a traversal state of this data structure.

Note
An `IlcIntSet` object should not contain an integer of value `IlcIntMax` . If an integer of value of `IlcIntMax` is included in an `IlcIntSet` object, an error can occur when iterating over the set.

See Also: IlcIntArray, IlcIntSetArray, IlcIntSetIterator, IlcIntSetVar, operator<<

Constructor Summary	
`public`	`IlcIntSet()`
`public`	`IlcIntSet(IlcIntSetI * impl)`
`public`	`IlcIntSet(IloSolver s, IlcInt min, IlcInt max, IlcBool fullSet=IlcTrue)`
`public`	`IlcIntSet(const IlcIntArray array, IlcBool fullSet=IlcTrue)`
`public`	`IlcIntSet(IloSolver s, const IlcIntArray array, IlcBool fullSet=IlcTrue)`

Method Summary	
`public IlcBool`	`add(IlcInt elt)`
`public IlcIntSet`	`copy() const`
`public IlcIntSetI *`	`getImpl() const`
`public IlcInt`	`getSize() const`
`public IloSolver`	`getSolver() const`
`public IlcBool`	`isIn(IlcInt elt) const`
`public void`	`operator=(const IlcIntSet & h)`
`public IlcBool`	`remove(IlcInt elt)`

Constructors

`public` **`IlcIntSet`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public IlcIntSet(IlcIntSetI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public IlcIntSet(IloSolver s, IlcInt min, IlcInt max, IlcBool fullSet=IlcTrue)`

This constructor creates a finite set of integers ranging from `min` to `max` included. If `min` is greater than `max`, then the empty set is created. If the argument `fullset` is equal to `IlcTrue`, the default value, the finite set will initially contain all its possible values. Otherwise, the finite set will initially be empty.

`public IlcIntSet(const IlcIntArray array, IlcBool fullSet=IlcTrue)`

This constructor also creates a finite set of integers containing the elements of `array`.

`public IlcIntSet(IloSolver s, const IlcIntArray array, IlcBool fullSet=IlcTrue)`

This constructor creates a finite set of integers containing the elements of `array`. If `array` contains multiple copies of a given value, that value will appear only one time in the newly created finite set. If the argument `fullset` is equal to `IlcTrue`, its default value, the finite set will initially contain all its possible values. Otherwise, the finite set will initially be empty. In any case, the possible elements of the finite set are exactly those elements in `array`.

Methods

`public IlcBool add(IlcInt elt)`

This member function adds `elt` to the invoking finite set if `elt` is a possible member of that set and if `elt` is not already in that set. When both conditions are met, this member function returns `IlcTrue`. Otherwise, it returns `IlcFalse`. The effects of this member function are reversible.

`public IlcIntSet copy() const`

This member function creates and returns a finite set that contains the same elements as the invoking finite set.

`public IlcIntSetI * getImpl() const`

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

`public IlcInt getSize() const`

This member function returns the size of a finite set. Clearly, this member function is useful for testing whether the invoking finite set is empty or not.

`public IloSolver **getSolver**() const`

This member function returns an instance of `IloSolver` associated with the invoking object.

`public IlcBool **isIn**(IlcInt elt) const`

This member function is a predicate that indicates whether or not `elt` is in the invoking finite set. It returns `IlcTrue` if `elt` is in the set; otherwise, it returns `IlcFalse`.

`public void **operator=**(const IlcIntSet & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

`public IlcBool **remove**(IlcInt elt)`

This member function removes `elt` from the invoking finite set. This member function returns `IlcFalse` if `elt` was not in that invoking set. Otherwise, it returns `IlcTrue`. The effects of this member function are reversible.

Class IlcIntSetArray

Definition file: ilsolver/intset.h
Include file: <ilsolver/ilosolver.h>

An instance of `IlcIntSetArray` represents an array of sets, instances of `IlcIntSet`.

For each basic type, Solver defines a corresponding array class. This array class is a handle class. In other words, an object of this class contains a pointer to another object allocated on the Solver heap. Exploiting handles in this way greatly simplifies the programming interface since the handle can then be an automatic object: as a developer using handles, you do not have to worry about memory allocation.

Empty Handle or Null Array

It is possible to create a null array, or in other words, an empty handle. When you do so, only these operations are allowed on that null array:

- **copy**: You can assign the null array to a new array.
- **access its size**: The member function `getSize` for the null array returns 0 (zero).
- **create an iterator**: You can create an iterator to traverse the null array. The member function `ok` returns `IlcFalse` for a null array.

Attempts to access a null array in any other way will throw an exception (an instance of `IloSolver::SolverErrorException`).

See Also: IlcIntSet, operator<<

Constructor Summary	
public	`IlcIntSetArray()`
public	`IlcIntSetArray(IlcIntSetArrayI * impl)`
public	`IlcIntSetArray(IloSolver solver, IlcInt size, IlcIntSet * values)`
public	`IlcIntSetArray(IloSolver solver, IlcInt size, IlcIntArray array)`
public	`IlcIntSetArray(IloSolver solver, IlcInt size, const IlcIntSet exp...)`

Method Summary	
`public IlcIntSetArrayI *`	`getImpl() const`
`public const char *`	`getName() const`
`public IlcInt`	`getSize() const`
`public IloSolver`	`getSolver() const`
`public IloSolverI *`	`getSolverI() const`
`public void`	`operator=(const IlcIntSetArray & h)`
`public IlcIntSet &`	`operator[](IlcInt i) const`
`public void`	`setName(const char * name) const`

Constructors

`public` **`IlcIntSetArray`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public IlcIntSetArray(IlcIntSetArrayI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public IlcIntSetArray(IloSolver solver, IlcInt size, IlcIntSet * values)`

This constructor creates an array of sets; the length of that array is `size`; its elements are initialized with the values indicated by `values`.

`public IlcIntSetArray(IloSolver solver, IlcInt size, IlcIntArray array)`

This constructor creates an array of sets; the length of that array is `size`; its elements are initialized with the values indicated by `array`.

`public IlcIntSetArray(IloSolver solver, IlcInt size, const IlcIntSet exp...)`

This constructor creates an array of sets; the length of that array is `size`; its elements are initialized with the arguments of type `IlcIntSet`. The number of arguments of type `IlcIntSet` must be the same as `size`.

Methods

`public IlcIntSetArrayI * getImpl() const`

This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

`public const char * getName() const`

This member function returns the name of the invoking object.

`public IlcInt getSize() const`

This member function returns the number of elements in the invoking array.

`public IloSolver getSolver() const`

This member function returns an instance of `IloSolver` associated with the invoking object.

`public IloSolverI * getSolverI() const`

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

`public void operator=(const IlcIntSetArray & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

`public IlcIntSet & operator[](IlcInt i) const`

This operator returns a reference to the element at rank `i`. This operator can be used for accessing (that is, simply reading) the element or for modifying (that is, writing) it.

`public void setName(const char * name) const`

This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

Class IlcIntSetIterator

Definition file: ilsolver/intexp.h
Include file: <ilsolver/ilosolver.h>

An instance of the class `IlcIntSetIterator` is an iterator that traverses the elements of finite sets of integers (instances of `IlcIntSet`).

For more information, see the concept Iterator.

See Also: IlcIntSet

Constructor Summary	
public	IlcIntSetIterator(IlcIntSet set)

Method Summary	
public IlcBool	ok() const
public IlcInt	operator*() const
public IlcIntSetIterator &	operator++()

Constructors

`public` **`IlcIntSetIterator`**`(IlcIntSet set)`

This constructor creates an iterator associated with `set` to traverse its elements.

Methods

`public IlcBool` **`ok`**`() const`

This member function returns `IlcTrue` if there is a current element and the invoking iterator points to it. Otherwise, it returns `IlcFalse`.

To traverse the elements of a finite set integers, use the following code:

```
IlcAny val;
for(IlcIntSetIterator iter(set); iter.ok(); ++iter){
              val = *iter;
              // do something with val
}
```

`public IlcInt` **`operator*`**`() const`

This operator returns the current element, the one to which the invoking iterator points.

`public IlcIntSetIterator &` **`operator++`**`()`

This operator advances the iterator to point to the next value in the set.

Class IlcIntSetSelect

Definition file: ilsolver/intset.h
Include file: <ilsolver/ilosolver.h>

Solver lets you control the order in which the values in the domain of a constrained set variable are tried during the search for a solution.

This class is the handle class of the object that chooses the value to try when the constrained set variable under consideration is a constrained integer set variable (that is, an instance of `IlcIntSetVar`).

An object of this handle class uses the undocumented virtual member function `IlcIntSetSelectI::select` from its implementation class to choose a value in the domain of the constrained integer variable under consideration during the search for a solution.

See the example in `IlcIntSelect` or the example in `IlcAnySetSelect` for a model of how to create and use a selector.

See Also: IlcEvalIntSet, IloInstantiate, IloIntSetValueSelector, IloIntSetValueSelectorI

Constructor Summary	
`public`	`IlcIntSetSelect()`
`public`	`IlcIntSetSelect(IlcIntSetSelectI * impl)`
`public`	`IlcIntSetSelect(IloSolver solver, IlcEvalIntSet function)`

Method Summary	
`public IlcIntSetSelectI *`	`getImpl() const`
`public void`	`operator=(const IlcIntSetSelect & h)`
`public IlcInt`	`select(IlcIntSetVar var) const`

Constructors

`public` **`IlcIntSetSelect`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcIntSetSelect`**`(IlcIntSetSelectI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public` **`IlcIntSetSelect`**`(IloSolver solver, IlcEvalIntSet function)`

This constructor creates a new integer set selector from an evaluation function. The implementation object of the newly created handle is an instance of the class `IlcIntSetSelectEvalI` constructed with the evaluation function indicated by the argument `function`.

Methods

```
public IlcIntSetSelectI * getImpl() const
```

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

```
public void operator=(const IlcIntSetSelect & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

```
public IlcInt select(IlcIntSetVar var) const
```

This member function uses the virtual member function of its implementation class to select a constrained integer set variable.

Class IlcIntSetVar

Definition file: ilsolver/intset.h
Include file: <ilsolver/ilosolver.h>

A constrained set variable is any instance of the class `IlcIntSetVar` or `IlcAnySetVar`. The value of a variable belonging to the class `IlcIntSetVar` is an instance of the class `IlcIntSet`. The value of a variable belonging to the class `IlcAnySetVar` is an instance of the class `IlcAnySet`. These two classes are handle classes. They both have the same implementation class, `IlcIntSetVarI`.

Domain

The domain associated with a constrained set variable is a *set of sets*. Solver represents this kind of domain by its upper and lower bounds. The upper bound is the union of all the possible values for the variable, that is the union of all the element-sets of the domain. The lower bound is the intersection of all the possible values of the variables, that is the intersection of all the element-sets of the domain.

In other words, the domain of a constrained set variable is represented by two sets:

- the *required set*, that is, the set of those elements that belong to all the possible values of the variable (the lower bound);
- the *possible set*, that is the set of those elements that belong to at least one of the possible values of the variable (the upper bound).

The possible set contains the required set by construction. The value, the possible set, and the required set of a constrained set variable are all instances of the classes `IlcAnySet` or `IlcIntSet`. When a constrained set variable is *bound*, the required elements are the same as the possible ones, and they are the elements of the *value* of the variable.

Delta Sets and Propagation

When a propagation event is triggered for a constrained set variable, the variable is pushed into the constraint propagation queue if it was not already in the queue. Moreover, the modifications of the domain of the constrained set variable are stored in two special sets. The first set stores the values removed from the possible set of the constrained set variable, and it is called the *possible-delta set*. The second one stores the values added to the required set of the constrained set variable, and it is called the *required-delta set*. These delta sets can be accessed during the propagation of the constraints posted on that variable. When all the constraints posted on that variable have been processed, then the delta sets are cleared. If the variable is modified again, then the whole process begins again. The state of the delta sets is reversible.

Failure

The domain of a constrained set variable can be reduced until it is empty, that is, to the point that the required set is not included in the possible set. In such a case, *failure* is triggered since at that point, no solution is possible.

Cardinality (Size of Set)

It is also possible to constrain the *cardinality* of the value of a constrained set variables. A constrained set variables contains a data member that is constrained integer variable (called the cardinality variable); it represents how many elements are in the value of the set variable. The minimum of the cardinality variable is always greater than or equal to the size of the required set. Its maximum is always less than or equal to the size of the possible set. The functions `IlcCard` (for sets and for indices) access cardinality.

Backtracking and Reversibility

All member functions defined for this class and capable of modifying constrained set variables are *reversible.* In particular, the changes made by constraint-posting functions are made with reversible assignments. Thus, the value, domain, and constraints posted on any constrained set variables are restored when Solver backtracks.

A modifier is a member function that reduces the domain of a constrained integer set variable, if it can. The modifier is not stored, in contrast to a constraint. If the domain becomes empty, a failure occurs. Modifiers are usually used to define new *classes* of constraints.

See Also: IlcCard, IlcCard, IlcIntersection, IlcIntSet, IlcIntSetIterator, IlcIntSetVarArray, IlcMember, IlcUnion, operator<<

Constructor Summary	
public	IlcIntSetVar()
public	IlcIntSetVar(IlcIntSetVarI * impl)
public	IlcIntSetVar(IloSolver solver, IlcInt min, IlcInt max, const char * name=0)
public	IlcIntSetVar(IloSolver solver, const IlcIntArray array, const char * name=0)

Method Summary	
public void	addRequired(IlcInt elt) const
public IlcIntSetVar	getCopy(IloSolver solver) const
public IlcIntSetVarI *	getImpl() const
public const char *	getName() const
public IlcAny	getObject() const
public IlcIntSet	getPossibleSet() const
public IlcIntSet	getRequiredSet() const
public IlcInt	getSize() const
public IloSolver	getSolver() const
public IloSolverI *	getSolverI() const
public IlcIntSet	getValue() const
public IlcBool	isBound() const
public IlcBool	isInDomain(IlcIntSet set) const
public IlcBool	isInProcess() const
public IlcBool	isPossible(IlcInt elt) const
public IlcBool	isRequired(IlcInt elt) const
public void	operator=(const IlcIntSetVar & h)
public void	removePossible(IlcInt elt) const
public void	setDomain(IlcIntSetVar var) const
public void	setName(const char * name) const
public void	setObject(IlcAny object) const
public void	whenDomain(IlcDemon demon) const
public void	whenValue(IlcDemon demon) const

Constructors

public **IlcIntSetVar**()

This constructor creates an empty handle. You must initialize it before you use it.

`public IlcIntSetVar(IlcIntSetVarI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public IlcIntSetVar(IloSolver solver, IlcInt min, IlcInt max, const char * name=0)`

This constructor creates a constrained set of integers with no required elements; its possible elements range from `min` to `max`, included. If `min` is greater than `max`, this constructor will throw an exception (an instance of `IloSolver::SolverErrorException`).

`public IlcIntSetVar(IloSolver solver, const IlcIntArray array, const char * name=0)`

This constructor creates a constrained set of integers with no required elements; its possible elements are the integers in `array`, an array of integers.

Methods

`public void addRequired(IlcInt elt) const`

The way this member function behaves depends on whether its argument `elt` is a member of the required or possible set of the invoking object. If `elt` is already in the required set of the invoking object, then this member function does nothing. If `elt` is in the possible set of the invoking object, but not yet in the required set, then this member function adds `elt` to the required set. If `elt` is not in the possible set of the invoking object, then this member function indicates a failure.

`public IlcIntSetVar getCopy(IloSolver solver) const`

This member function returns a copy of the invoking constrained set variable and associates that copy with `solver`.

`public IlcIntSetVarI * getImpl() const`

This constructor creates an object by copying another one. This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

`public const char * getName() const`

This member function returns the name of the invoking object.

`public IlcAny getObject() const`

This member function returns a pointer to the external object associated with the invoking object, if there is such an association. It returns 0 (zero) otherwise.

`public IlcIntSet getPossibleSet() const`

This member function returns the possible set of the invoking constrained set variable

`public IlcIntSet getRequiredSet() const`

This member function returns the required set of the invoking constrained set variable.

`public IlcInt **getSize**() const`

This member function returns one plus the difference between the cardinality of the set of possible elements and the cardinality of the set of required elements. Don't confuse the size of the *domain* of the constrained set variable (returned by this member function) with the cardinality of the set to which the variable is *bound* (that is, the size of the *value* of the variable). The cardinality of the set variable itself is a constrained integer variable returned by the function `IlcCard`.

`public IloSolver **getSolver**() const`

This member function returns an instance of `IloSolver` associated with the invoking object.

`public IloSolverI * **getSolverI**() const`

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

`public IlcIntSet **getValue**() const`

This member function returns the value of the invoking constrained set variable if that variable has been bound; otherwise, it will throw an exception (an instance of `IloSolver::SolverErrorException`).

`public IlcBool **isBound**() const`

This member function returns `IlcTrue` if the constrained set variable has been bound, that is, if its set of required elements is equal to its set of possible elements. Otherwise, the member function returns `IlcFalse`.

`public IlcBool **isInDomain**(IlcIntSet set) const`

This member function returns `IlcTrue` if and only if the finite set indicated by `set` satisfies the following conditions:

- The set contains all the required elements of the invoking constrained set variable.
- Each element of `set` is a possible element of the invoking constrained set variable.

`public IlcBool **isInProcess**() const`

This member function returns `IlcTrue` if the invoking constrained set variable is currently being processed by the constraint propagation algorithm. Only one variable can be in process at a time.

`public IlcBool **isPossible**(IlcInt elt) const`

This member function returns `IlcTrue` if `elt` is a possible element in the invoking constrained set variable. It returns `IlcFalse` otherwise. This member function could be defined as `getPossibleSet().isIn(elt)`.

`public IlcBool **isRequired**(IlcInt elt) const`

This member function returns `IlcTrue` if `elt` is a required element in the invoking constrained set variable. It returns `IlcFalse` otherwise. This member function could be defined as `getRequiredSet().isIn(elt)`.

`public void **operator=**(const IlcIntSetVar & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

`public void **removePossible**(IlcInt elt) const`

The way this member function behaves depends on whether its argument `elt` is a member of the required set of the invoking object. If `elt` is in the required set of the invoking object, then this member function indicates a failure. If `elt` is in the possible set of the invoking object, but not in the required set, then this member function removes `elt` from the possible set.

`public void **setDomain**(IlcIntSetVar var) const`

This member function reduces the domain of the invoking constrained set variable so that its domain becomes included in the domain of the constrained set variable `var`.

If the invoking variable is already bound, then this member function considers whether its value belongs to the domain of `var`. If its value does *not* belong to the domain of `var`, then the member function indicates failure.

If the invoking variable is not yet bound, then its required set is replaced by its union with the required set of `var`, and its possible set is replaced by its intersection with the possible set of `var`. If the resulting required set is not included in the resulting possible set, then the member function indicates failure. If the resulting required set contains the same elements as the resulting possible set, then the invoking variable is bound to that remaining value. In any case, if the invoking variable is modified, the constraints posted on it are activated.

The effects of this member function are reversible.

`public void **setName**(const char * name) const`

This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

`public void **setObject**(IlcAny object) const`

This member function establishes a link between the invoking object and an external object of which the invoking object might be a data member.

`public void **whenDomain**(IlcDemon demon) const`

This member function associates `demon` with the domain propagation event of the invoking constrained set variable. Whenever the domain of the invoking constrained set variable changes, the demon will be executed immediately.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. Whenever the domain of the invoking constrained set variable changes, the constraint will be propagated.

`public void **whenValue**(IlcDemon demon) const`

This member function associates `demon` with the value propagation event of the invoking constrained set variable. Whenever the invoking constrained set variable becomes bound, the demon will be executed immediately.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. Whenever the invoking constrained set variable becomes bound, the constraint will be propagated.

Class IlcIntSetVarArray

Definition file: ilsolver/intset.h
Include file: <ilsolver/ilosolver.h>

An *array* of constrained set variables of integers is an instance of the class `IlcIntSetVarArray`. The elements of such an array are instances of `IlcIntSetVar`.

Empty Handle or Null Array

It is possible to create a null array, or in other words, an empty handle. When you do so, only these operations are allowed on that null array:

- **copy**: You can assign the null array to a new array.
- **access its size**: The member function `getSize` for the null array returns 0 (zero).
- **create an iterator**: You can create an iterator to traverse the null array. The member function `ok` returns `IlcFalse` for a null array.

Attempts to access a null array in any other way will throw an exception (an instance of `IloSolver::SolverErrorException`).

See Also: IlcIntSetVar, IlcIntSetVarArrayIterator, operator<<

Constructor Summary	
public	IlcIntSetVarArray()
public	IlcIntSetVarArray(IlcIntSetVarArrayI * impl)
public	IlcIntSetVarArray(IloSolver solver, IlcInt size)
public	IlcIntSetVarArray(IloSolver solver, IlcInt size, IlcInt min, IlcInt max)
public	IlcIntSetVarArray(IloSolver s, IlcInt size, IlcIntArray array)
public	IlcIntSetVarArray(IloSolver solver, IlcInt size ILCPARAM, const IlcIntSetVar v1, const IlcIntSetVar v2)

Method Summary	
public IlcIntSetVarArray	getCopy(IloSolver solver) const
public IlcIntSetVarArrayI *	getImpl() const
public const char *	getName() const
public IloSolver	getSolver() const
public IloSolverI *	getSolverI() const
public void	operator=(const IlcIntSetVarArray & h)
public IlcIntSetVar &	operator[](IlcInt index) const
public void	setName(const char * name) const

Constructors

`public` **`IlcIntSetVarArray`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

```
public IlcIntSetVarArray(IlcIntSetVarArrayI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IlcIntSetVarArray(IloSolver solver, IlcInt size)
```

This constructor creates an array of `size` uninitialized constrained set variables. The argument `size` must be strictly greater than 0 (zero). The index range of the array is `[0 size)`, where 0 is included but `size` is excluded. Each element of the array must be assigned a value before the array can be used.

```
public IlcIntSetVarArray(IloSolver solver, IlcInt size, IlcInt min, IlcInt max)
```

This constructor creates an array of `size` constrained set variables. The index range of the array is `[0 size)` where 0 is included, but `size` is excluded. The argument `size` must be strictly greater than 0 (zero). None of the constrained variables has any required elements; for each of the constrained variables, the possible elements range from `min` to `max`, included.

```
public IlcIntSetVarArray(IloSolver s, IlcInt size, IlcIntArray array)
```

This constructor creates an array of `size` constrained set variables. The argument `size` must be strictly greater than 0 (zero). The index range of the array is `[0 size)`, where 0 is included but `size` is excluded. Each constrained variable has no required elements; its possible elements are the pointers in `array`, an array of pointers.

```
public IlcIntSetVarArray(IloSolver solver, IlcInt size ILCPARAM, const IlcIntSetVar
v1, const IlcIntSetVar v2)
```

This constructor creates an array of `size` elements. The elements must be successive. If `size` is different from the number of instances of `IlcIntSetVar` passed to the constructor, then the behavior of this constructor is undefined and unlikely to be what you want. The argument `size` must be strictly greater than 0 (zero).

Methods

```
public IlcIntSetVarArray getCopy(IloSolver solver) const
```

This member function returns a copy of the invoking array of constrained set variables and associates that copy with `solver`.

```
public IlcIntSetVarArrayI * getImpl() const
```

This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

```
public const char * getName() const
```

This member function returns the name of the invoking object.

```
public IloSolver getSolver() const
```

This member function returns an instance of `IloSolver` associated with the invoking object.

```
public IloSolverI * getSolverI() const
```

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

```
public void operator=(const IlcIntSetVarArray & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

```
public IlcIntSetVar & operator[](IlcInt index) const
```

This subscripting operator returns a reference to the variable at rank `index` in the invoking array. The fact the operator returns a reference makes it possible for you, as a developer, to assign the corresponding constrained expression.

```
public void setName(const char * name) const
```

This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

Class IlcIntSetVarArrayIterator

Definition file: ilsolver/intset.h
Include file: <ilsolver/ilosolver.h>

Instances of the class `IlcIntSetVarArrayIterator` traverse the values of an array of *sets* of constrained integer variables.

For more information, see the concept Iterator.

See Also: IlcIntSetVar, IlcIntSetVarArray

Constructor and Destructor Summary	
`public`	`IlcIntSetVarArrayIterator(const IlcIntSetVarArray array)`

Method Summary	
`public IlcBool`	`next(IlcIntSetVar & variable)`

Constructors and Destructors

`public` **`IlcIntSetVarArrayIterator`**`(const IlcIntSetVarArray array)`

This constructor creates an iterator to traverse the values belonging to an array of sets of constrained integer variables. This iterator lets you iterate forward over the complete index range of the array.

Methods

`public IlcBool` **`next`**`(IlcIntSetVar & variable)`

This member function takes a reference to a constrained integer set variable and returns a Boolean value. It begins with the first element. It returns `IlcFalse` if there is no other element on which to iterate and `IlcTrue` otherwise. When it returns `IlcTrue`, it writes the next element of the iterator (forward iteration) to the argument.

Class IlcIntToFloatExpFunction

Definition file: ilsolver/accessor.h
Include file: <ilsolver/ilosolver.h>

It is sometimes useful to associate a constrained variable with an element of the domain of a constrained variable. If the elements of a domain are objects, the associated values can correspond to a specific constrained attribute of these objects.

The following constraints and expressions use this kind of indirection: `IlcSum`, `IlcMin`, `IlcMax`, and `IlcUnion`.

`IlcIntToFloatExpFunction` is the handle class of the object that makes the correspondence between an integer element and a constrained floating-point expression or variable.

An object of this handle class uses the virtual member function IlcIntToFloatExpFunctionI::getValue from its implementation class to obtain the associated constrained variable or expression of an element of a domain.

See Also: IlcIntToFloatExpFunctionI

Constructor Summary	
`public`	`IlcIntToFloatExpFunction()`
`public`	`IlcIntToFloatExpFunction(IlcIntToFloatExpFunctionI * impl)`

Method Summary	
`public IlcIntToFloatExpFunctionI *`	`getImpl() const`
`public IlcFloatExp`	`getValue(IlcInt i) const`
`public void`	`operator=(const IlcIntToFloatExpFunction & h)`

Constructors

`public` **`IlcIntToFloatExpFunction`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcIntToFloatExpFunction`**`(IlcIntToFloatExpFunctionI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

Methods

`public IlcIntToFloatExpFunctionI *` **`getImpl`**`() const`

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

```
public IlcFloatExp getValue(IlcInt i) const
```

This member function returns the constrained floating-point variable or expression associated with the integer element `i`.

```
public void operator=(const IlcIntToFloatExpFunction & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IlcIntToFloatExpFunctionI

Definition file: ilsolver/accessi.h
Include file: <ilsolver/ilsolver.h>

This class is the implementation class for IlcIntToFloatExpFunction, the class of the object that makes the association between an integer value and a constrained floating-point variable or expression.

The virtual member function `getValue` returns the constrained variable or expression associated with a given value. To define an association class, you have to derive this class, and overload the `getValue` virtual member function.

The following constraints and expressions use this kind of indirection: `IlcSum`, `IlcMin`, `IlcMax`, and `IlcUnion`.

See Also: IlcIntToFloatExpFunction

Constructor and Destructor Summary	
public	IlcIntToFloatExpFunctionI()
public	~IlcIntToFloatExpFunctionI()

Method Summary	
public virtual IlcFloatExp	getValue(IlcInt e)

Constructors and Destructors

public **IlcIntToFloatExpFunctionI**()

This constructor creates an implementation object.

public **~IlcIntToFloatExpFunctionI**()

As this class is to be subclassed, a virtual destructor is provided

Methods

public virtual IlcFloatExp **getValue**(IlcInt e)

This member function must return the constrained floating-point variable or expression associated with the integer element `e`.

Class IlcIntToIntExpFunction

Definition file: ilsolver/accessor.h
Include file: <ilsolver/ilosolver.h>

It is sometimes useful to associate a constrained variable with an element of the domain of a constrained variable. If the elements of a domain are objects, the associated values can correspond to a specific constrained attribute of these objects.

The following constraints and expressions use this kind of indirection: `IlcSum`, `IlcMin`, `IlcMax`, and `IlcUnion`.

`IlcIntToIntExpFunction` is the handle class of the object that makes the correspondence between an integer element and a constrained integer expression or variable.

An object of this handle class uses the virtual member function IlcIntToIntExpFunctionI::getValue from its implementation class to obtain the associated constrained variable or expression of an element of a domain.

See Also: IlcIntToIntExpFunctionI

Constructor Summary	
public	IlcIntToIntExpFunction()
public	IlcIntToIntExpFunction(IlcIntToIntExpFunctionI * impl)

Method Summary	
public IlcIntToIntExpFunctionI *	getImpl() const
public IlcIntExp	getValue(IlcInt i) const
public void	operator=(const IlcIntToIntExpFunction & h)

Constructors

public **IlcIntToIntExpFunction**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IlcIntToIntExpFunction**(IlcIntToIntExpFunctionI * impl)

This constructor creates a handle object from a pointer to an implementation object.

Methods

public IlcIntToIntExpFunctionI * **getImpl**() const

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

`public IlcIntExp` **`getValue`**`(IlcInt i) const`

This member function returns the constrained integer variable or expression associated with the integer element `i`.

`public void` **`operator=`**`(const IlcIntToIntExpFunction & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IlcIntToIntExpFunctionI

Definition file: ilsolver/accessi.h
Include file: <ilsolver/ilsolver.h>

This class is the implementation class for IlcIntToIntExpFunction, the class of the object that makes the association between an integer value and a constrained integer variable or expression.

The virtual member function `getValue` returns the constrained variable or expression associated with a given value. To define an association class, you have to derive this class, and overload the `getValue` virtual member function.

The following constraints and expressions use this kind of indirection: `IlcSum`, `IlcMin`, `IlcMax`, and `IlcUnion`.

See Also: IlcIntToIntExpFunction, IlcAnyToIntExpFunction, IlcAnyToIntFunction

Constructor and Destructor Summary	
public	IlcIntToIntExpFunctionI()
public	~IlcIntToIntExpFunctionI()

Method Summary	
public virtual IlcIntExp	getValue(IlcInt e)

Constructors and Destructors

public **IlcIntToIntExpFunctionI**()

This constructor creates an implementation object.

public **~IlcIntToIntExpFunctionI**()

As this class is to be subclassed, a virtual destructor is provided

Methods

public virtual IlcIntExp **getValue**(IlcInt e)

This member function must return the constrained variable or expression associated with the integer element `e`.

Class IlcIntToIntStepFunction

Definition file: ilsolver/segfunc.h
Include file: <ilsolver/ilosolver.h>

An instance of `IlcIntToIntStepFunction` represents a step function over integers which is defined everywhere on an interval `[xmin,xmax)`. Each interval `[x1,x2)` over which the function has the same value is called a step. The member functions of the class `IlcIntToIntStepFunction` are not reversible.

See Also: IlcIntToIntStepFunctionCursor, IlcMax, IlcMin, operator+, operator-, operator-, operator*, operator==, operator<=, operator>=

Constructor Summary	
public	`IlcIntToIntStepFunction()`
public	`IlcIntToIntStepFunction(IlcSegmentedFunctionI * impl)`
public	`IlcIntToIntStepFunction(IloSolver solver, IlcInt xmin=IlcIntMin, IlcInt xmax=IlcIntMax, IlcInt dval=0)`
public	`IlcIntToIntStepFunction(IloSolver solver, IlcIntArray x, IlcIntArray v, IlcInt xmin=IlcIntMin, IlcInt xmax=IlcIntMax)`

Method Summary	
public void	`addValue(IlcInt x1, IlcInt x2, IlcInt v)`
public void	`dilate(IlcInt k)`
public IlcInt	`getArea(IlcInt x1, IlcInt x2) const`
public IlcInt	`getDefinitionIntervalMax() const`
public IlcInt	`getDefinitionIntervalMin() const`
public IlcSegmentedFunctionI *	`getImpl() const`
public IlcInt	`getMax(IlcInt x1, IlcInt x2) const`
public IlcInt	`getMin(IlcInt x1, IlcInt x2) const`
public const char *	`getName() const`
public IlcAny	`getObject() const`
public IloSolver	`getSolver() const`
public IloSolverI *	`getSolverI() const`
public IlcInt	`getValue(IlcInt x) const`
public void	`operator*=(IlcInt k)`
public void	`operator+=(const IlcIntToIntStepFunction & fct)`
public void	`operator-=(const IlcIntToIntStepFunction & fct)`
public void	`operator=(const IlcIntToIntStepFunction & h)`
public void	`setMax(const IlcIntToIntStepFunction & fct)`
public void	`setMax(IlcInt x1, IlcInt x2, IlcInt v)`
public void	`setMin(const IlcIntToIntStepFunction & fct)`
public void	`setMin(IlcInt x1, IlcInt x2, IlcInt v)`
public void	`setName(const char * name) const`

public void	setObject(IlcAny object) const
public void	setPeriodic(const IlcIntToIntStepFunction & fp, IlcInt x0, IlcInt n=IlcIntMax, IlcInt dval=0)
public void	setSteps(IlcIntArray x, IlcIntArray v)
public void	setValue(IlcInt x1, IlcInt x2, IlcInt v)
public void	shift(IlcInt dx, IlcInt dval=0)

Constructors

`public` **`IlcIntToIntStepFunction`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcIntToIntStepFunction`**`(IlcSegmentedFunctionI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public` **`IlcIntToIntStepFunction`**`(IloSolver solver, IlcInt xmin=IlcIntMin, IlcInt xmax=IlcIntMax, IlcInt dval=0)`

This constructor creates an integer step function defined everywhere on the interval `[xmin,xmax)` with the same value `dval`.

`public` **`IlcIntToIntStepFunction`**`(IloSolver solver, IlcIntArray x, IlcIntArray v, IlcInt xmin=IlcIntMin, IlcInt xmax=IlcIntMax)`

This constructor creates an integer step function defined everywhere on the interval `[xmin,xmax);` its steps are defined by the two arrays of parameters, `x` and `v`.

More precisely, if `n` is the size of array `x`, the size of array `v` must be `n+1` and, if the created function is defined on the interval `[xmin,xmax)`, its values will be:

- `v[0]` on interval `[xmin,x[0])`,
- `v[i]` on interval `[x[i-1],x[i])` for all `i` in `[0,n-1]`, and
- `v[n]` on interval `[x[n-1],xmax)`.

Methods

`public void` **`addValue`**`(IlcInt x1, IlcInt x2, IlcInt v)`

This member function increases the value of the invoking integer step function by `v` everywhere on the interval `[x1,x2)`.

`public void` **`dilate`**`(IlcInt k)`

This member function multiplies the scale of `x` by `k` for the invoking integer step function. The parameter `k` must be a positive integer.

More precisely, if the invoking function was defined over an interval `[xmin,xmax)`, it will be redefined over the interval `[k*xmin,k*xmax)` and the value at `x` will be the former value at `x/k`.

`public IlcInt getArea(IlcInt x1, IlcInt x2) const`

This member function returns the sum of the invoking integer step function over the interval `[x1,x2)`.

If the interval `[x1,x2)` is not included in the interval of definition of the invoking function, Solver will throw an exception (an instance of `IloSolver::SolverErrorException`).

`public IlcInt getDefinitionIntervalMax() const`

This member function returns the rightmost point of the interval of definition of the invoking step function.

`public IlcInt getDefinitionIntervalMin() const`

This member function returns the leftmost point of the interval of definition of the invoking step function.

`public IlcSegmentedFunctionI * getImpl() const`

This constructor creates an object by copying another one. This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

`public IlcInt getMax(IlcInt x1, IlcInt x2) const`

This member function returns the maximal value of the invoking integer step function on the interval `[x1,x2)`. If the interval `[x1,x2)` is not included in the interval of definition of the invoking function, Solver will throw an exception (an instance of `IloSolver::SolverErrorException`).

`public IlcInt getMin(IlcInt x1, IlcInt x2) const`

This member function returns the minimal value of the invoking integer step function on the interval `[x1,x2)`. If the interval `[x1,x2)` is not included in the interval of definition of the invoking function, Solver will throw an exception (an instance of `IloSolver::SolverErrorException`).

`public const char * getName() const`

This member function returns the name of the invoking object.

`public IlcAny getObject() const`

This member function returns a pointer to the external object associated with the invoking object, if there is such an association. It returns 0 (zero) otherwise.

`public IloSolver getSolver() const`

This member function returns an instance of `IloSolver` associated with the invoking object.

`public IloSolverI * getSolverI() const`

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

`public IlcInt getValue(IlcInt x) const`

This member function returns the value of the invoking integer step function at `x`. If `x` does not belong to the interval of definition of the invoking function, Solver will throw an exception (an instance of `IloSolver::SolverErrorException`).

`public void operator*=(IlcInt k)`

This operator multiplies the value of the invoking integer step function by a factor `k` everywhere on the interval of definition.

`public void operator+=(const IlcIntToIntStepFunction & fct)`

This operator adds the parameter function `fct` to the invoking integer step function.

`public void operator-=(const IlcIntToIntStepFunction & fct)`

This operator subtracts the parameter function `fct` from the invoking integer step function.

`public void operator=(const IlcIntToIntStepFunction & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

`public void setMax(const IlcIntToIntStepFunction & fct)`

This member function sets the value of the invoking integer step function to be the maximum between the current value and the value of `fct` everywhere on the interval of definition of the invoking function. The interval of definition of `fct` must be the same as that of the invoking step function.

`public void setMax(IlcInt x1, IlcInt x2, IlcInt v)`

This member function sets the value of the invoking integer step function to be the maximum between the current value and `v` everywhere on the interval `[x1,x2)`.

`public void setMin(const IlcIntToIntStepFunction & fct)`

This member function sets the value of the invoking integer step function to be the minimum between the current value and the value of fct everywhere on the interval of definition of the invoking function. The interval of definition of fct must be the same as the one of the invoking step function.

`public void setMin(IlcInt x1, IlcInt x2, IlcInt v)`

This member function sets the value of the invoking integer step function to be the minimum between the current value and `v` everywhere on the interval `[x1,x2)`.

`public void **setName**(const char * name) const`

This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

`public void **setObject**(IlcAny object) const`

This member function establishes a link between the invoking object and an external object of which the invoking object might be a data member.

`public void **setPeriodic**(const IlcIntToIntStepFunction & fp, IlcInt x0, IlcInt n=IlcIntMax, IlcInt dval=0)`

This member function initializes the invoking function as an integer step function that repeats the step function `fp`, `n` times after `x0`.

More precisely, if `fp` is defined on `[xfpmin,xfpmax)` and if the invoking function is defined on `[xmin,xmax)`, the value of the invoking function will be:

- `dval` on `[xmin, x0)`,
- `fp((x-x0) % (xfpmax-xfpmin))` for `x` in `[x0, Min(x0+n*(xfpmax-xfpmin), xmax))`, and
- `dval` on `[Min(x0+n*(xfpmax-xfpmin), xmax), xmax)`

`public void **setSteps**(IlcIntArray x, IlcIntArray v)`

This member function initializes the invoking function as an integer step function whose steps are defined by the two parameters arrays `x` and `v`.

More precisely, if `n` is the size of array `x`, size of array `v` must be `n+1` and, if the invoking function is defined on the interval `[xmin,xmax)`, its values will be:

- `v[0]` on interval `[xmin,x[0])`,
- `v[i]` on interval `[x[i-1],x[i])` for all `i` in `[0,n-1]`, and
- `v[n]` on interval `[x[n-1],xmax)`.

`public void **setValue**(IlcInt x1, IlcInt x2, IlcInt v)`

This member function sets the value of the invoking integer step function to be `v` on the interval `[x1,x2)`.

`public void **shift**(IlcInt dx, IlcInt dval=0)`

This member function shifts the invoking function from `dx` to the right if `dx > 0` or from `-dx` to the left if `dx < 0`. It has no effect if `dx = 0`.

More precisely, if the invoking function is defined on `[xmin,xmax)` and `dx > 0`, the new value of the invoking function is:

- `dval` on the interval `[xmin,xmin+dx)`,
- for all `x` in `[xmin+dx,xmax)`: former value at `x-dx`.

If `dx < 0`, the new value of the invoking function is:

- for all `x` in `[xmin,xmax+dx)`: former value at `x-dx`,
- `dval` on the interval `[xmax+dx,xmax)`.

Class IlcIntToIntStepFunctionCursor

Definition file: ilsolver/segfunc.h
Include file: <ilsolver/ilosolver.h>

An instance of the class `IlcIntToIntStepFunctionCursor` allows you to inspect the contents of an integer step function. A segment of an integer step function is defined as an interval `[x1, x2)` over which the value of the function is the same. Cursors iterate forward or backward over the segments of an integer step function.

> **Note**
>
> The structure of the integer step function must not be changed while a cursor is inspecting it. Therefore functions that change the structure of the step function, such as `IlcIntToIntStepFunction::setValue` should not be called while a cursor is in use.

See Also: IlcIntToIntStepFunction

Constructor and Destructor Summary	
public	IlcIntToIntStepFunctionCursor(const IlcIntToIntStepFunction &, IlcInt x)

Method Summary	
public IlcInt	getSegmentMax() const
public IlcInt	getSegmentMin() const
public IlcInt	getValue() const
public IlcBool	ok() const
public void	operator++()
public void	operator--()

Constructors and Destructors

`public` **`IlcIntToIntStepFunctionCursor`**`(const IlcIntToIntStepFunction &, IlcInt x)`

This constructor creates a cursor to inspect the integer step function argument. This cursor lets you iterate forward or backward over the segments of the function. The cursor initially indicates the segment of the function that contains `x`.

Methods

`public IlcInt` **`getSegmentMax`**`() const`

This member function returns the rightmost point of the segment currently indicated by the cursor.

`public IlcInt` **`getSegmentMin`**`() const`

This member function returns the leftmost point of the segment currently indicated by the cursor.

`public IlcInt **getValue**() const`

This member function returns the value of the segment currently indicated by the cursor.

`public IlcBool **ok**() const`

This member function returns `IlcFalse` if the cursor does not currently indicate a segment included in the interval of definition of the integer step function. Otherwise, it returns `IlcTrue`.

An attempt to use the cursor after `ok()` returns `IlcFalse` leads to undefined behavior.

`public void **operator++**()`

This operator moves the cursor to the segment adjacent to the current segment (forward move).

`public void **operator--**()`

This operator moves the cursor to the segment adjacent to the current segment (backward move).

Class IlcIntTupleSet

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

An integer tuple is an ordered set of integer values. A *set* of integer tuples is represented by an instance of `IlcIntTupleSet`. That is, the elements of an integer tuple *set* are integer tuples. The number of values in a tuple is known as the *arity* of the tuple, and the arity of the tuples in a set is called the *arity* of the set. (In contrast, the number of tuples in the set is known as the *cardinality* of the set.)

As a handle class, `IlcIntTupleSet` manages certain set operations efficiently. In particular, elements can be added to such a set. It is also possible to search a given set with the member function `IlcIntTupleSet::isIn` to see whether or not the set contains a given element.

In addition, a set of integer tuples can represent a constraint defined on a constrained integer variable, either as the set of *allowed* combinations of values of the constrained variable on which the constraint is defined, or as the set of *forbidden* combinations of values.

There are a few conventions governing integer tuple sets:

- When you create the set, you must specify the arity of the tuple-elements it contains.
- You use the member function `IlcIntTupleSet::add` to add integer tuples to the set.
- Before searching to determine whether or not an integer tuple belongs to a given set, you must *close* the set by calling the member function `IlcIntTupleSet::close`. This operation—-closing the set—-improves the performance of certain other operations on the set.

Solver will throw an exception (an instance of `IloSolver::SolverErrorException`) if you attempt:

- to add an integer tuple with a different number of variables from the arity of the set;
- to add an element to a set that has already been closed;
- to search for an integer tuple with an arity different from the set arity;
- to search for an integer tuple in a set that has not been closed yet;
- to define a constraint on an integer tuple set that has not been closed already.

You do not have to worry about memory allocation. If you respect these conventions, Solver manages allocation and de-allocation transparently for you.

See Also: IlcTableConstraint

Constructor Summary	
`public`	`IlcIntTupleSet()`
`public`	`IlcIntTupleSet(IlcECSetOfSharedTupleI * impl)`
`public`	`IlcIntTupleSet(IloSolver solver, IlcInt arity)`

Method Summary	
`public void`	`add(IlcIntArray tuple) const`
`public void`	`close() const`
`public IlcECSetOfSharedTupleI *`	`getImpl() const`
`public IlcBool`	`isClosed() const`
`public IlcBool`	`isIn(IlcIntArray tuple) const`
`public void`	`operator=(const IlcIntTupleSet & h)`

public void	setBigTuple() const
public void	setHoloTuple() const
public void	setSimpleTuple() const

Constructors

`public **IlcIntTupleSet**()`

This constructor creates an empty handle. You must initialize it before you use it.

`public **IlcIntTupleSet**(IlcECSetOfSharedTupleI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public **IlcIntTupleSet**(IloSolver solver, IlcInt arity)`

This constructor creates a set of integer tuples (an instance of the class `IlcIntTupleSet`) with the arity indicated by `arity`.

Methods

`public void **add**(IlcIntArray tuple) const`

This member function adds an integer tuple represented by the array `tuple` to the invoking set. If you attempt to add an element that is already in the set, that element will *not* be added again; in other words, Solver respects the definition of a set as containing exclusive elements with no duplicates. Added elements are not copied; that is, there is no memory duplication. Solver will throw an exception (an instance of `IloSolver::SolverErrorException`) if the set has already been closed. Solver will throw an exception (an instance of `IloSolver::SolverErrorException`) if the size of the array is not equal to the arity of the invoking set.

`public void **close**() const`

This member function closes the invoking set. That is, it states that all the tuples in the set are known so efficient data structures can be defined and exploited.

`public IlcECSetOfSharedTupleI * **getImpl**() const`

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

`public IlcBool **isClosed**() const`

This member function returns `IlcTrue` if the invoking set has been closed. Otherwise, it returns `IlcFalse`.

`public IlcBool` **`isIn`**`(IlcIntArray tuple) const`

This member function returns `IlcTrue` if tuple belongs to the invoking set. Otherwise, it returns `IlcFalse`. Solver will throw an exception (an instance of `IloSolver::SolverErrorException`) if the set has not yet been closed. Solver will throw an exception (an instance of `IloSolver::SolverErrorException`) if the size of the array is not equal to the arity of the invoking set.

`public void` **`operator=`**`(const IlcIntTupleSet & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

`public void` **`setBigTuple`**`() const`

This member function states that the tuples in the set will be compile in a specific data structure. It must be called before close().

`public void` **`setHoloTuple`**`() const`

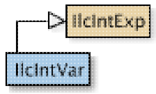
This member function states that the tuples in the set will be compile in a specific data structure. It must be called before close().

`public void` **`setSimpleTuple`**`() const`

This member function states that the tuples in the set will be compile in a specific data structure (the one by default). It must be called before close().

Class IlcIntVar

Definition file: ilsolver/intexp.h
Include file: <ilsolver/ilosolver.h>



In a typical application exploiting Solver, the unknowns of the problem will be expressed as constrained variables. The most commonly used class of constrained variables is the class of constrained *integer* variables. `IlcIntVar` is one of a group of classes for expressing constraints on constrained integer variables. In fact, `IlcIntVar` (the class of constrained integer variables) is a subclass of IlcIntExp, the class of constrained integer expressions. A constrained integer variable is a constrained integer expression whose domain is explicitly stored.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Domain

The domain of a constrained integer expression is *computed* from the domains of its subexpressions. For example, the domain of the expression `x+y` contains the range `[x.getMin()+y.getMin(), x.getMax()+y.getMax()]`.

A constrained integer variable is a constrained expression that *stores* its domain instead of computing it from its subexpressions. The domain of a constrained integer variable contains values of type `IlcInt`. This domain is represented by an interval when the values are consecutive or by an enumeration of integers otherwise.

You can convert a constrained integer expression (which computes its domain) into a constrained integer variable (which stores its domain) by either of two means: by the casting constructor or by the assignment operator.

Domain-Delta and Propagation

When a propagation event is triggered for a constrained variable, the variable is pushed into the propagation queue if it was not already in the queue. Moreover, the modifications of the domain of the constrained variable are stored in a special set called the *domain-delta*. This domain-delta can be accessed during the propagation of the constraints posted on that variable. When all the constraints posted on that variable have been processed, then the domain-delta is cleared. If the variable is modified again, then the whole process begins again. The state of the domain-delta is reversible.

Backtracking and Reversibility

All the member functions and operators defined for this class and capable of modifying constrained variables are *reversible*. In particular, the changes made by constraint-posting functions are made with reversible assignments. Thus, the value, the domain, and the constraints posted on any constrained variable are restored when Solver backtracks.

See Also: IlcIntExp, IlcIntExpIterator, IlcIntVarArray, IlcIntVarDeltaIterator, IlcTableConstraint

Constructor Summary	
`public`	`IlcIntVar()`
`public`	`IlcIntVar(IloSolver s, IlcInt min, IlcInt max, const char * name=0)`
`public`	`IlcIntVar(IloSolver s, const IlcIntArray values, const char * name=0)`
`public`	`IlcIntVar(IlcIntVarI * impl)`
`public`	`IlcIntVar(const IlcIntExp exp)`

Method Summary	
public IlcInt	getMax() const
public IlcInt	getMaxDelta() const
public IlcInt	getMin() const
public IlcInt	getMinDelta() const
public IlcInt	getOldMax() const
public IlcInt	getOldMin() const
public IlcBool	isInDelta(IlcInt value) const
public IlcBool	isInProcess() const
public void	operator=(const IlcIntVar & exp)
public void	operator=(const IlcIntExp & exp)

Inherited Methods from `IlcIntExp`
getCopy, getImpl, getMax, getMin, getName, getNextHigher, getNextLower, getObject, getSize, getSolver, getSolverI, getValue, isBound, isInDomain, operator=, removeDomain, removeDomain, removeRange, removeValue, setDomain, setDomain, setDomain, setMax, setMin, setName, setObject, setRange, setValue, whenDomain, whenRange, whenValue

Constructors

public **IlcIntVar**()

This constructor creates a constrained integer variable which is empty, that is, whose handle pointer is null. This object must then be assigned before it can be used, exactly as when you, as a developer, declare a pointer.

public **IlcIntVar**(IloSolver s, IlcInt min, IlcInt max, const char * name=0)

This constructor creates a constrained integer variable with a domain containing all the integer values between `min` and `max`, inclusive. If `min` is greater than `max`, the function `IlcFail` is called. The optional argument `name`, if provided, becomes the name of the constrained integer variable.

For example, if we want to create a constrained integer variable with the domain `[0 ... 10]`, then we use the constructor like this:

```
IlcIntVar var (s, 0, 10);
```

public **IlcIntVar**(IloSolver s, const IlcIntArray values, const char * name=0)

In case you want to create a constrained integer variable where the domain is not an interval of integers, this constructor creates a constrained integer variable with a domain containing exactly those integers that belong to `values`, an array of integers. The optional argument `name`, if provided, becomes the name of the constrained integer variable.

Here's how to use that constructor to create a constrained integer variable with a domain of non-consecutive integers.

```
IlcIntArray values (s, 10, 28, 45, 65, 78, 90, 102, 113,
```

```
                   123, 123, 138);
 IlcIntVar aVar (values);
```

`public IlcIntVar(IlcIntVarI * impl)`

This constructor creates a handle object (an instance of the class `IlcIntVar` from a pointer to an object (an instance of the class `IlcIntVarI`.

`public IlcIntVar(const IlcIntExp exp)`

To transform a constrained integer *expression* (which computes its domain from its subexpressions) into a constrained integer *variable* (which stores its domain), you can use this constructor. It associates a domain with the constrained integer expression `exp`. This expression thus becomes a constrained integer variable. Moreover, the newly created constrained integer variable points to the same implementation object as `exp`. (You can also use the assignment operator for the same purpose.)

Methods

`public IlcInt getMax() const`

This member function returns the maximum of the domain of the invoking object.

`public IlcInt getMaxDelta() const`

This member function returns the difference between the maximum of the domain of the invoking constrained variable and the maximum of its domain-delta. This member function can be applied only to the variable currently in process.

For example, to know whether the maximum of a constrained integer variable `x` has been modified since the last time the constraints posted on `x` were processed, it is sufficient to test the value of `x.getMaxDelta()`. If that test returns 0, then the maximum of `x` has not been modified.

`public IlcInt getMin() const`

This member function returns the minimum of the domain of the invoking object.

`public IlcInt getMinDelta() const`

This member function returns the difference between the minimum of the domain of the invoking constrained variable and the minimum of its domain-delta. This member function can be applied only to the variable currently in process.

For example, to know whether the minimum of a constrained integer variable `x` has been modified since the last time the constraints posted on `x` were processed, it is sufficient to test the value of `x.getMinDelta()`. If that test returns 0, then the minimum of `x` has not been modified.

`public IlcInt getOldMax() const`

This member function returns the maximum of the domain-delta of the invoking constrained variable. This member function can be applied only to the variable currently in process.

`public IlcInt` **`getOldMin`**`() const`

This member function returns the minimum of the domain-delta of the invoking constrained variable. This member function can be applied only to the variable currently in process.

`public IlcBool` **`isInDelta`**`(IlcInt value) const`

This member function returns `IlcTrue` if the argument `value` belongs to the domain-delta of the invoking constrained variable. This member function can be applied only to the variable currently in process.

`public IlcBool` **`isInProcess`**`() const`

This member function returns `IlcTrue` if the invoking constrained variable is currently being processed by the constraint propagation algorithm. Only one variable can be in process at a time.

`public void` **`operator=`**`(const IlcIntVar & exp)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the argument `exp`. After the execution of this operator, the invoking object and the `exp` object both point to the same implementation object. This assignment operator has no effect on its argument.

`public void` **`operator=`**`(const IlcIntExp & exp)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IlcIntVarArray

Definition file: ilsolver/intexp.h
Include file: <ilsolver/ilosolver.h>

The class `IlcIntVarArray` is the class for an *array* of instances of `IlcIntVar`. Three integers—`indexMin`, `indexMax`, and `indexStep`—play an important role in such an array of constrained integer variables. The index of those variables ranges from `indexMin`, inclusive, to `indexMax`, exclusive, in steps of `indexStep`. The index of the first variable in the array is `indexMin`; the second one is `indexMin+indexStep`, and so forth. The quantity `indexMax-indexMin` must be a multiple of `indexStep`.

Generic Constraints

The array makes it easier to implement *generic* constraints. In this context, a generic constraint is a constraint that applies to all of the variables in the array. Member functions of the array class are available to post such generic constraints. A generic constraint is then allocated and recorded only once for all the variables in the array. This fact represents a significant economy in memory, compared to allocating and recording one constraint per variable.

Interval Constraints

Arrays of constrained variables also allow you to define *interval constraints* which propagate in a global way when the domains of one or more constrained variables in the array are modified. Propagation is then performed through a goal. Member functions such as `whenValueInterval`, `whenRangeInterval`, or `whenDomainInterval` associate goals with propagation events for this purpose.

Backtracking and Reversibility

All the functions and member functions capable of modifying arrays of constrained integer variables are *reversible*. In particular, when modifiers and functions that post constraints are called, the state before their call will be saved by Solver.

Empty Handle or Null Array

It is possible to create a null array, or in other words, an empty handle. When you do so, only these operations are allowed on that null array:

- **copy**: You can assign the null array to a new array.
- **access its size**: The member function `getSize` for the null array returns 0 (zero).
- **create an iterator**: You can create an iterator to traverse the null array. The member function `ok` returns `IlcFalse` for a null array.

Attempts to access a null array in any other way will throw an exception (an instance of `IloSolver::SolverErrorException`).

See Also: IlcAbstraction, IlcIndex, IlcIntVar, IlcIntVarArrayIterator, operator<<

Constructor Summary	
public	IlcIntVarArray()
public	IlcIntVarArray(IlcIntVarArrayI * impl)
public	IlcIntVarArray(IloSolver solver, IlcInt size)
public	IlcIntVarArray(IloSolver solver, IlcInt size ILCPARAM, const IlcIntVar v1, const IlcIntVar v2)
public	IlcIntVarArray(IloSolver solver, IlcInt size, IlcInt min, IlcInt max)

public	IlcIntVarArray(IloSolver s, IlcInt indexMin, IlcInt indexMax, IlcInt indexStep, const IlcIntVar prototype)

Method Summary	
public IlcIntVarArray	getCopy(IloSolver solver) const
public IlcInt	getDomainIndexMax() const
public IlcInt	getDomainIndexMin() const
public IlcIntVarArrayI *	getImpl() const
public IlcInt	getIndexMax() const
public IlcInt	getIndexMin() const
public IlcInt	getIndexStep() const
public IlcInt	getIndexValue() const
public IlcInt	getMaxMax() const
public IlcInt	getMaxMax(IlcInt indexMin, IlcInt indexMax) const
public IlcInt	getMaxMin() const
public IlcInt	getMaxMin(IlcInt indexMin, IlcInt indexMax) const
public IlcInt	getMinMax() const
public IlcInt	getMinMax(IlcInt indexMin, IlcInt indexMax) const
public IlcInt	getMinMin() const
public IlcInt	getMinMin(IlcInt indexMin, IlcInt indexMax) const
public const char *	getName() const
public IlcInt	getRangeIndexMax() const
public IlcInt	getRangeIndexMin() const
public IlcInt	getSize() const
public IloSolver	getSolver() const
public IloSolverI *	getSolverI() const
public IlcInt	getValueIndexMax() const
public IlcInt	getValueIndexMin() const
public IlcIntVar	getVariable(IlcInt index, IlcBool before=IlcFalse) const
public void	operator=(const IlcIntVarArray & h)
public IlcIntExp	operator[](const IlcIntExp exp) const
public IlcIntExp	operator[](IlcIndex & I) const
public IlcIntVar &	operator[](IlcInt index) const
public void	setName(const char * name) const
public void	whenDomain(const IlcDemon demon)
public void	whenDomainInterval(const IlcDemon demon)
public void	whenRange(const IlcDemon demon)
public void	whenRangeInterval(const IlcDemon demon)
public void	whenValue(const IlcDemon demon)
public void	whenValueInterval(const IlcDemon demon)

Constructors

`public IlcIntVarArray()`

This constructor creates an empty handle. You must initialize it before you use it.

`public IlcIntVarArray(IlcIntVarArrayI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public IlcIntVarArray(IloSolver solver, IlcInt size)`

This constructor creates an uninitialized array of length `size`. The argument `size` must be strictly greater than 0 (zero). The index range of the array is `[0 size)`. Each element of the array must be assigned before the array can be used.

`public IlcIntVarArray(IloSolver solver, IlcInt size ILCPARAM, const IlcIntVar v1, const IlcIntVar v2)`

This constructor creates an array of length `size`. Its constrained integer variables are initialized with the list of variables provided as arguments to the constructor. The number of `IlcIntVar` arguments must be equal to `size`. The argument `size` must be strictly greater than 0 (zero).

`public IlcIntVarArray(IloSolver solver, IlcInt size, IlcInt min, IlcInt max)`

This constructor creates an array of `size` constrained integer variables. The argument `size` must be strictly greater than 0 (zero). Each constrained integer variable has a domain containing all integer values between `min` and `max`.

`public IlcIntVarArray(IloSolver s, IlcInt indexMin, IlcInt indexMax, IlcInt indexStep, const IlcIntVar prototype)`

This constructor creates an array of copies of the given constrained integer variable `prototype`. Each copy initially has the same domain as `prototype` currently has, but the copies do not share the constraints of `prototype`. That is, they are independent of it.

The index range of the array is `[indexMin indexMax)`. The step of the array is `indexStep`. Solver will throw an exception (an instance of `IloSolver::SolverErrorException`) if any of the following conditions occur:

- `indexMin` is not strictly less than `indexMax`;
- `indexStep` is not strictly positive;
- `(indexMax-indexMin)` is not a multiple of `indexStep`.

This constructor keeps no pointer to the `prototype` variable. That variable may be an automatic object allocated on the C++ stack.

Methods

`public IlcIntVarArray getCopy(IloSolver solver) const`

This member function returns a copy of the invoking array of constrained variables and associates that copy with `solver`.

`public IlcInt getDomainIndexMax() const`

When it is called during the execution of a goal associated with an array by the member function `whenDomainInterval`, this member function returns the maximum of the range of the array `[indexMin indexMax)` over which some removal of values has occurred. The returned value of this member function is not meaningful outside the execution of a goal associated with the array by the member function `whenDomainInterval`.

`public IlcInt getDomainIndexMin() const`

When it is called during the execution of a goal associated with an array by the member function `whenDomainInterval`, this member function returns the minimum of the range of the array `[indexMin indexMax)` over which some removal of values has occurred. The returned value of this member function is not meaningful outside the execution of a goal associated with the array by the member function `whenDomainInterval`.

`public IlcIntVarArrayI * getImpl() const`

This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

`public IlcInt getIndexMax() const`

This member function returns the maximal index of the invoking array of constrained integer variables.

`public IlcInt getIndexMin() const`

This member function returns the minimal index of the invoking array of constrained integer variables.

`public IlcInt getIndexStep() const`

This member function returns the index step of the invoking array of constrained integer variables. The meaning of this index step is that the indexed variable value may change only at indices equal to `(getIndexMin() + i * getIndexStep())`.

`public IlcInt getIndexValue() const`

When it is called during the execution of a constraint or demon associated with an array by the member functions `whenValue, whenDomain,` or `whenRange`, this member function returns the index in the invoking array of the constrained variable that triggered the propagation event. Calling this member function outside the execution of the goal will throw an exception (an instance of `IloSolver::SolverErrorException`) with the message "`unbound index`".

`public IlcInt getMaxMax() const`

This member function returns the largest of the maximal values of the variables belonging to the invoking array of constrained integer variables.

`public IlcInt getMaxMax(IlcInt indexMin, IlcInt indexMax) const`

This member function returns the largest of the maximal values of the variables belonging to the invoking array of constrained integer variables. The arguments `indexMin` and `indexMax` are provided, only those variables that correspond to indices in the range `[indexMin indexMax)` are considered. Solver will throw an exception (an instance of `IloSolver::SolverErrorException` with the message `"bad index interval"` if the given `indexMin` and `indexMax` are not valid indices for the invoking array of constrained variables or if `indexMin` is not strictly less than `indexMax`.

`public IlcInt getMaxMin() const`

This member function returns the largest of the minimal values of the variables belonging to the invoking array of constrained integer variables.

`public IlcInt getMaxMin(IlcInt indexMin, IlcInt indexMax) const`

This member function returns the largest of the minimal values of the variables belonging to the invoking array of constrained integer variables. The arguments `indexMin` and `indexMax` are provided, only those variables that correspond to indices in the range `[indexMin indexMax)` are considered. Solver will throw an exception (an instance of `IloSolver::SolverErrorException` with the message `"bad index interval"` if the given `indexMin` and `indexMax` are not valid indices for the invoking array of constrained variables or if `indexMin` is not strictly less than `indexMax`.

`public IlcInt getMinMax() const`

This member function returns the smallest of the maximal values of the variables belonging to the invoking array of constrained integer variables.

`public IlcInt getMinMax(IlcInt indexMin, IlcInt indexMax) const`

This member function returns the smallest of the maximal values of the variables belonging to the invoking array of constrained integer variables. The arguments `indexMin` and `indexMax` are provided, only those variables that correspond to indices in the range `[indexMin indexMax)` are considered. Solver will throw an exception (an instance of `IloSolver::SolverErrorException` with the message `"bad index interval"` if the given `indexMin` and `indexMax` are not valid indices for the invoking array of constrained variables or if `indexMin` is not strictly less than `indexMax`.

`public IlcInt getMinMin() const`

This member function returns the smallest of the minimal values of the variables belonging to the invoking array of constrained integer variables.

`public IlcInt getMinMin(IlcInt indexMin, IlcInt indexMax) const`

This member function returns the smallest of the minimal values of the variables belonging to the invoking array of constrained integer variables. The arguments `indexMin` and `indexMax` are provided, only those variables that correspond to indices in the range `[indexMin indexMax)` are considered. Solver will throw an exception (an instance of `IloSolver::SolverErrorException` with the message `"bad index interval"` if the given `indexMin` and `indexMax` are not valid indices for the invoking array of constrained variables or if `indexMin` is not strictly less than `indexMax`.

`public const char * `**`getName`**`() const`

This member function returns the name of the invoking object.

`public IlcInt `**`getRangeIndexMax`**`() const`

When it is called during the execution of a goal associated with an array by the member function `whenRangeInterval`, this member function returns the maximum of the range of the array `[indexMin indexMax)` over which some removal of values has occurred. The returned value of this member function is not meaningful outside the execution of a goal associated with the array by the member function `whenRangeInterval`.

`public IlcInt `**`getRangeIndexMin`**`() const`

When it is called during the execution of a goal associated with an array by the member function `whenRangeInterval`, this member function returns the minimum of the range of the array `[indexMin indexMax)` over which some removal of values has occurred. The returned value of this member function is not meaningful outside the execution of a goal associated with the array by the member function `whenRangeInterval`.

`public IlcInt `**`getSize`**`() const`

This member function returns the number of variables in the invoking array.

`public IloSolver `**`getSolver`**`() const`

This member function returns an instance of `IloSolver` associated with the invoking object.

`public IloSolverI * `**`getSolverI`**`() const`

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

`public IlcInt `**`getValueIndexMax`**`() const`

When it is called during the execution of a goal associated with an array by the member function `whenValueInterval`, this member function returns the maximum of the range of the array `[indexMin indexMax)` over which some binding has occurred. The returned value of this member function is not meaningful outside the execution of a goal associated with the array by the member function `whenValueInterval`.

`public IlcInt `**`getValueIndexMin`**`() const`

When it is called during the execution of a goal associated with an array by the member function `whenValueInterval`, this member function returns the minimum of the range of the array `[indexMin` indexMax) over which some binding has occurred. The returned value of this member function is not meaningful outside the execution of a goal associated with the array by the member function whenValueInterval.

public IlcIntVar **getVariable**(IlcInt index, IlcBool before=IlcFalse) const

This member function returns the variable corresponding to the given index in the invoking array of constrained integer variables. However, if before is IlcTrue, then getVariable returns the variable before the variable at the given index. Solver will throw an exception (an instance of IloSolver::SolverErrorException) with the message "bad index" if the given index is not a valid one for the invoking array of constrained integer variables.

If t is an array of constrained integer variables, then these three expressions return the same value:

```
t.getVariable(index, IlcFalse)
t.getVariable(index + 1, IlcTrue)
t[index]
```

public void **operator=**(const IlcIntVarArray & h)

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

public IlcIntExp **operator[]**(const IlcIntExp exp) const

This subscripting operator returns a constrained integer expression. For clarity, let's call A the invoking array. When exp is bound to the value i, then the domain of the expression is the domain of A[i]. More generally, the domain of the expression is the union of the domains of the expressions A[i] where the i are in the domain of exp.

public IlcIntExp **operator[]**(IlcIndex & I) const

This operator returns a generic variable, which is a constrained variable representing an element of the array. That generic variable is said to *stem from* the index i.

public IlcIntVar & **operator[]**(IlcInt index) const

This subscripting operator returns a reference to a constrained integer variable corresponding to the given index in the invoking array of constrained integer variables. Solver will throw an exception (an instance of IloSolver::SolverErrorException) with the message "bad index" if the given index is not a valid one for the invoking array of constrained integer variables.

public void **setName**(const char * name) const

This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

public void **whenDomain**(const IlcDemon demon)

This member function associates demon with the domain propagation event of every variable in the invoking array. Whenever a value is removed from the domain of any of the variables in the invoking array, the demon will be executed immediately.

When the demon is executed, the index of the constrained variable that triggered the domain event can be known by a call to the member function `getIndexValue`.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. Whenever a value is removed from the domain of any of the variables in the invoking array, the constraint will be propagated.

`public void` **`whenDomainInterval`**`(const IlcDemon demon)`

This member function associates `demon` with the domain propagation event of every variable in the invoking array. It specifies that a given demon reacts globally to modifications of the domains of a collection of variables in the array. Whenever a domain propagation event or a series of such events occurs, the demon will be executed immediately.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. Whenever a domain propagation event or a series of such events occurs, the constraint will be propagated.

A call to the demon signifies that *some* removal of values from domain(s) occurred over the index range `[indexMin indexMax)`. It does *not* mean that values have been removed from the domains of all the variables in the range.

`public void` **`whenRange`**`(const IlcDemon demon)`

This member function associates `demon` with the range propagation event of every variable in the invoking array. Whenever any bound of any of the variables in the array is modified, the demon will be executed immediately.

When the demon is executed, the index of the constrained integer variable that has triggered the range event can be known by a call to the member function `getIndexValue`.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. Whenever any bound of any of the variables in the array is modified, the constraint will be propagated.

`public void` **`whenRangeInterval`**`(const IlcDemon demon)`

This member function associates `demon` with the range propagation event of every variable in the invoking array. It specifies that a given demon reacts globally to modifications of the boundaries of a collection of variables in the array. Whenever a range propagation event or a series of such events occurs, the demon will be executed immediately.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. Whenever a range propagation event or a series of such events occurs, the constraint will be propagated.

A call to the demon signifies that *some* modification of the boundaries occurred to variables over the index range `[indexMin indexMax)`. It does *not* mean that all the variables in the range had their boundaries modified.

`public void` **`whenValue`**`(const IlcDemon demon)`

This member function associates `demon` with the value propagation event of every variable in the invoking array. When one of the variables in the array receives a value, the demon will be executed immediately.

When the demon is executed, the index of the bound constrained variable can be known by a call to the member function `getIndexValue`.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. When one of the variables in the array receives a value, the constraint will be propagated.

```
public void whenValueInterval(const IlcDemon demon)
```

This member function associates `demon` with the value propagation event of every variable in the invoking array. It specifies that a given demon reacts globally to the binding of a collection of variables in the array. When a value propagation event or a series of such events occurs, the demon will be executed immediately.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. When a value propagation event or a series of such events occurs, the constraint will be propagated.

A call to the demon signifies that *some* variable binding occurred over the index range `[indexMin indexMax)`. It does *not* mean that all the variables in the range have been bound.

Class IlcIntVarArrayIterator

Definition file: ilsolver/intexp.h
Include file: <ilsolver/ilosolver.h>

An instance of the class `IlcIntVarArrayIterator` traverse the values of an array of constrained integer variables.

For more information, see the concept Iterator.

See Also: IlcIntVar, IlcIntVarArray

Constructor and Destructor Summary	
`public`	`IlcIntVarArrayIterator(const IlcIntVarArray array)`

Method Summary	
`public IlcBool`	`next(IlcIntVar & variable)`

Constructors and Destructors

`public` **`IlcIntVarArrayIterator`**`(const IlcIntVarArray array)`

This constructor creates an iterator to traverse the values belonging to the array of constrained integer variables. This iterator lets you iterate forward over the complete index range of the array.

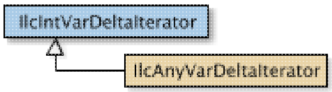
Methods

`public IlcBool` **`next`**`(IlcIntVar & variable)`

This member function takes a reference to a constrained integer variable and returns a Boolean value. It returns `IlcFalse` if there is no other element on which to iterate and `IlcTrue` otherwise. When it returns `IlcTrue`, it writes the next element of the iterator (forward iteration) to the argument.

Class IlcIntVarDeltaIterator

Definition file: ilsolver/intexp.h
Include file: <ilsolver/ilosolver.h>



An instance of the class `IlcIntVarDeltaIterator` is an iterator that traverses the values belonging to the domain-delta of a constrained integer variable (that is, an instance of `IlcIntVar`).

For more information, see the concepts Propagation, Domain-Delta, and Iterator.

See Also: IlcIntVar

Constructor and Destructor Summary	
public	IlcIntVarDeltaIterator(const IlcIntVar var)

Method Summary	
public IlcBool	ok() const
public IlcInt	operator*() const
public IlcIntVarDeltaIterator &	operator++()

Constructors and Destructors

public **IlcIntVarDeltaIterator**(const IlcIntVar var)

This constructor creates an iterator associated with `var` to traverse the values belonging to the domain-delta of `var`.

Methods

public IlcBool **ok**() const

This member function returns `IlcTrue` if there is a current element and the iterator points to it. Otherwise, it returns `IlcFalse`.

To traverse the values belonging to the domain-delta of a constrained integer variable, use the following code:

```
IlcInt val;
for (IlcIntVarDeltaIterator iter(var); iter.ok(); ++iter){
        val = *iter;
        // do something with val
}
```

public IlcInt **operator***() const

This operator returns the current element, the one to which the invoking iterator points.

```
public IlcIntVarDeltaIterator & operator++()
```

This operator advances the iterator to point to the next value in the domain-delta of the constrained integer variable.

Class IlcMemoryManagerI

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>

An instance of this abstract implementation class is a memory manager. Its purpose is to delete automatically any memory used by a solver (an instance of `IloSolver`) but not allocated on the Solver heap.

In other words, if your application systematically allocates memory for a solver but does not use reversible allocation (by using `IloSolver::getHeap`) to do so, yet you want that memory to be deleted automatically, you should add a memory manager to the invoking solver to delete that allocated memory.

Use the member function `IloSolver::addMemoryManager` to add a memory manager to a solver. That member function will add the memory manager to the list of memory mangers associated with the invoking solver. Then, when the member function `IloEnv::end` is called for the invoking solver or when the solver re-extracts the model during synchronization, the virtual member function `IlcMemoryManagerI::end` will be called for all the memory managers on that list.

`IlcMemoryManagerI` is an abstract implementation class, so every time you define a subclass of this abstract class, you must redefine the virtual member function `IlcMemoryManagerI::end` to delete allocated memory appropriately.

Here is how we recommend that you use this class.

1. Define your own class.

```
class MyMemoryManager : public IlcMemoryManagerI{
     void end();
   };
  void MyMemoryManager::end() {
 }
```

2. At run time, use your new class like this:

```
s.addMemoryManager(new (s.getImpl()) MyMemoryManager());
```

See Also: IloSolver

Constructor and Destructor Summary	
`public`	`IlcMemoryManagerI()`

Method Summary	
`protected virtual void`	`end()`

Constructors and Destructors

`public` **`IlcMemoryManagerI`**`()`

This constructor creates a memory manager.

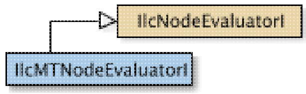
Methods

`protected virtual void end()`

This member function automatically deletes memory not allocated on the Solver heap. It must be redefined appopriately whenever you derive a subclass of `IlcMemoryManagerI`.

Class IlcMTNodeEvaluatorI

Definition file: ilsolver/search.h
Include file: <ilsolver/ilosolver.h>



This implementation class derives from the abstract implementation class `IlcNodeEvaluatorI`. Instances of `IlcMTNodeEvaluatorI` are implementation objects of node evaluators for multithreaded search.

Handle and Implementation Classes

A node evaluator is an object in Concert Technology. Like other Concert Technology entities, a node evaluator is implemented by means of two classes: a handle class and an implementation class. In other words, an instance of the class `IlcNodeEvaluator` (a handle) contains a data member (the handle pointer) that points to an instance of the class `IlcNodeEvaluatorI` (its implementation object).

Purpose of a Node Evaluator

A node evaluator is linked to the life cycle of an open node. When a node is created, the member function `evaluate` is called to evaluate the node. When the solver has to decide whether it should jump to another node, it calls the member function `subsume`.

A node evaluator has its own life cycle:

- To be used internally, it must be cloned by Solver code using the member function `duplicate`.
- When the node evaluator is activated, the member function `init` is called automatically by Solver code.

Implementing Your Own Node Evaluator

You implement a node evaluator much the same way as you implement a search selector.

The function `IlcMTMinimizeVar` returns an instance of `IlcMTSearchSelector`. In the documentation of that function, we describe part of its implementation details. From that description, you see how a search selector is used and what you must include in your own implementation of a search selector.

To get an idea of how to implement your own node evaluator, see `IlcMTMinimizeVar`.

See Also: IlcFloatVarRef, IlcIntVarRef

Constructor Summary	
public	`IlcMTNodeEvaluatorI(IloSolver s, IlcBool duplicate)`

Method Summary	
public void	`addFloatVar(IlcFloatVar v, const char * name)`
public void	`addIntVar(IlcIntVar v, const char * name)`
public IlcFloatVar	`getFloatVar(IloSolver s, IlcFloatVarRef ref) const`
public IlcFloatVarRef	`getFloatVarRef(const char *) const`
public IlcIntVar	`getIntVar(IloSolver s, IlcIntVarRef ref) const`
public IlcIntVarRef	`getIntVarRef(const char * name) const`

Inherited Methods from **IlcNodeEvaluatorI**
duplicateEvaluator, evaluate, init, subsume

Constructors

public **IlcMTNodeEvaluatorI**(IloSolver s, IlcBool duplicate)

This constructor creates a node evaluator for multithreaded search. The parameter duplicate indicates whether the object is an internal copy used by the solver (duplicate = IlcTrue), or whether it is just a template.

Methods

public void **addFloatVar**(IlcFloatVar v, const char * name)

This member function stores the constrained floating-point variable v under the name indicated by name.

public void **addIntVar**(IlcIntVar v, const char * name)

This member function stores the constrained integer variable v under the name indicated by name.

public IlcFloatVar **getFloatVar**(IloSolver s, IlcFloatVarRef ref) const

This member function retrieves the variable stored by the solver s under the name corresponding to the reference ref.

public IlcFloatVarRef **getFloatVarRef**(const char *) const

This member function retrieves all the constrained floating-point variables stored under the name indicated by name.

public IlcIntVar **getIntVar**(IloSolver s, IlcIntVarRef ref) const

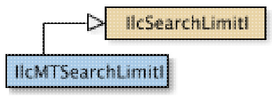
This member function retrieves the variable stored by the solver s under the name corresponding to the reference ref.

public IlcIntVarRef **getIntVarRef**(const char * name) const

This member function retrieves all the constrained integer variables stored under the name indicated by name.

Class IlcMTSearchLimitI

Definition file: ilsolver/search.h
Include file: <ilsolver/ilosolver.h>



This implementation class derives from the abstract implementation class `IlcSearchLimitI`. Instances of `IlcMTSearchLimitI` are implementation objects of search limits for multithreaded search.

Handle and Implementation Classes

A search limit is an object in Concert Technology. Like other Concert Technology entities, a search limit is implemented by means of two classes: a handle class and an implementation class. In other words, an instance of the class `IlcMTSearchLimit` (a handle) contains a data member (the handle pointer) that points to an instance of the class `IlcMTSearchLimitI` (its implementation object).

Purpose of a Search Limit

A search limit is used to prune part of the search tree. The member function `check` indicates whether the limit has been reached.

A search limit has its own life cycle:

- To be used internally by Concert Technology code, it has to be cloned by the member function `duplicate`.
- When it is activated, the member function `init` is called automatically by Concert Technology code.

Implementing Your Own Search Limit

You implement a search limit much the same way as you implement a search selector.

The function `IlcMTMinimizeVar` returns an instance of `IlcMTSearchSelector`. In the documentation of that function, we describe part of its implementation details. From that description, you see how a search selector is used and what you must include in your own implementation of a search selector.

To get an idea of how to implement your own search limit, see `IlcMTMinimizeVar`.

See Also: IlcMTSearchLimitI, IlcIntVarRef

Constructor Summary	
`public`	`IlcMTSearchLimitI(IloSolver s, IlcBool duplicate)`

Method Summary	
`public void`	`addFloatVar(IlcFloatVar v, const char * name)`
`public void`	`addIntVar(IlcIntVar v, const char * name)`
`public IlcFloatVar`	`getFloatVar(IloSolver s, IlcFloatVarRef ref) const`
`public IlcFloatVarRef`	`getFloatVarRef(const char *) const`
`public IlcIntVar`	`getIntVar(IloSolver s, IlcIntVarRef ref) const`
`public IlcIntVarRef`	`getIntVarRef(const char * name) const`

Inherited Methods from `IlcSearchLimitI`

```
check, duplicateLimit, init
```

Constructors

`public IlcMTSearchLimitI(IloSolver s, IlcBool duplicate)`

This constructor creates a search limit for multithreaded search. The parameter `duplicate` indicates whether the object is an internal copy used by the solver (`duplicate = IlcTrue`), or whether it is just a template.

Methods

`public void addFloatVar(IlcFloatVar v, const char * name)`

This member function stores the constrained floating-point variable `v` under the name indicated by `name`.

`public void addIntVar(IlcIntVar v, const char * name)`

This member function stores the constrained integer variable `v` under the name indicated by `name`.

`public IlcFloatVar getFloatVar(IloSolver s, IlcFloatVarRef ref) const`

This member function retrieves the variable stored by the solver `s` under the name corresponding to the reference `ref`.

`public IlcFloatVarRef getFloatVarRef(const char *) const`

This member function retrieves all the constrained floating-point variables stored under the name indicated by `name`.

`public IlcIntVar getIntVar(IloSolver s, IlcIntVarRef ref) const`

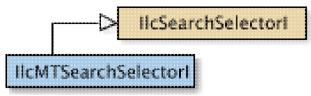
This member function retrieves the variable stored by the solver `s` under the name corresponding to the reference `ref`.

`public IlcIntVarRef getIntVarRef(const char * name) const`

This member function retrieves all the constrained integer variables stored under the name indicated by `name`.

Class IlcMTSearchSelectorI

Definition file: ilsolver/search.h
Include file: <ilsolver/search.h>



This implementation class derives from the abstract implementation class `IlcSearchSelectorI`. Instances of `IlcMTSearchSelectorI` are implementation objects of search selectors for *multithreaded* search.

Handle and Implementation Classes

A search selector is an object in Concert Technology. Like other Concert Technology entities, a search selector is implemented by means of two classes: a handle class and an implementation class. In other words, an instance of the class `IlcMTSearchSelector` (a handle) contains a data member (the handle pointer) that points to an instance of the class `IlcMTSearchSelectorI` (its implementation object).

Purpose of Search Selector

A search selector has several purposes when it is used internally by Concert Technology code:

- To implement a minimization process by the member functions `update`, `updateTo`, and `saveObjectiveValue`.
- To implement a filter on open nodes by the member functions `evaluate` and `isFeasible`.
- To manage *selected* nodes by the member functions `registerSolution`, `whenFinished`, `getCurrentNode`, `activateNode`, and `closeNode`.

A search selector has its own life cycle. To be used internally by Concert Technology code, it has to be cloned by the `duplicate` member function.

Implementing Your Own Search Selector

The function `IlcMTMinimizeVar` returns an instance of `IlcMTSearchSelector`. In the documentation of that function, we describe part of its implementation details. From that description, you can get an idea of how a search selector is used and what you must include in your own implementation of a search selector. See `IlcMTMinimizeVar`.

See Also: IlcFloatVarRef, IlcIntVarRef, IlcMTMinimizeVar

Constructor Summary	
public	`IlcMTSearchSelectorI(IloSolver s, IlcBool duplicate)`

Method Summary	
public void	`addFloatVar(IlcFloatVar v, const char * name)`
public void	`addIntVar(IlcIntVar v, const char * name)`
public IlcFloatVar	`getFloatVar(IloSolver s, IlcFloatVarRef ref) const`
public IlcFloatVarRef	`getFloatVarRef(const char *) const`
public IlcIntVar	`getIntVar(IloSolver s, IlcIntVarRef ref) const`
public IlcIntVarRef	`getIntVarRef(const char * name) const`

Inherited Methods from `IlcSearchSelectorI`

activateNode, closeNode, duplicateSelector, evaluate, getCurrentNode, isFeasible, saveObjectiveValue, updateObjective, updateObjectiveTo, whenFinishedTree, whenLeaf

Constructors

`public **IlcMTSearchSelectorI**(IloSolver s, IlcBool duplicate)`

This constructor creates a search selector for multithreaded search. The parameter `duplicate` indicates whether the object is an internal copy used by the solver (`duplicate = IlcTrue`), or whether it is just a template.

Methods

`public void **addFloatVar**(IlcFloatVar v, const char * name)`

This member function stores the constrained floating-point variable `v` under the name indicated by `name`.

`public void **addIntVar**(IlcIntVar v, const char * name)`

This member function stores the constrained integer variable `v` under the name indicated by `name`.

`public IlcFloatVar **getFloatVar**(IloSolver s, IlcFloatVarRef ref) const`

This member function retrieves the variable stored by the solver `s` under the name corresponding to the reference `ref`.

`public IlcFloatVarRef **getFloatVarRef**(const char *) const`

This member function retrieves all the constrained floating-point variables stored under the name indicated by `name`.

`public IlcIntVar **getIntVar**(IloSolver s, IlcIntVarRef ref) const`

This member function retrieves the variable stored by the solver `s` under the name corresponding to the reference `ref`.

`public IlcIntVarRef **getIntVarRef**(const char * name) const`

This member function retrieves all the constrained integer variables stored under the name indicated by `name`.

Class IlcNeighborIdentifier

Definition file: ilsolver/iimnhood.h
Include file: <ilsolver/ilosolver.h>

This class communicates information between instances of local search goals. This class is used by `IloScanNHood` and `IloScanDeltas` to store information about the currently instantiated neighbor. This class is used by `IloTest` and `IloNotify` to find out the details of the currently instantiated neighbor. This information can be communicated by passing the same instance of `IlcNeighborIdentifier` to such goals in a composed goal.

See Also: IloScanNHood, IloScanDeltas, IloTest, IloNotify, IloSingleMove, IloSolver, IloNeighborIdentifier, IloIIM

Constructor Summary	
public	IlcNeighborIdentifier()
public	IlcNeighborIdentifier(IlcNeighborIdentifierI * impl)
public	IlcNeighborIdentifier(IloSolver solver)

Method Summary	
public IloSolution	getAtDelta() const
public IloInt	getAtIndex() const
public IlcNeighborIdentifierI *	getImpl() const
public void	operator=(const IlcNeighborIdentifier & h)
public void	setAtDelta(IloSolution delta)
public void	setAtIndex(IloInt index)

Constructors

public **IlcNeighborIdentifier**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IlcNeighborIdentifier**(IlcNeighborIdentifierI * impl)

This constructor creates a handle object from a pointer to an implementation object.

public **IlcNeighborIdentifier**(IloSolver solver)

This constructor creates an instance of `IlcNeighborIdentifier` allocated on `solver`.

Methods

public IloSolution **getAtDelta**() const

This member function returns the solution passed in the previous call to `IlcNeighborIdentifier::setAtDelta`. This set is carried out automatically by `IloScanNHood` and `IloScanDeltas` before succeeding. The solution returned by this method represents the solution delta that was applied by `IloScanNHood` or `IloScanDeltas` to reach the current leaf node.

`public IloInt **getAtIndex**() const`

This member function returns the index passed in the previous call to `IlcNeighborIdentifier::setAtIndex`. This set is carried out automatically by `IloScanNHood` and `IloScanDeltas` before succeeding. The index returned by this method represents the neighborhood index (for `IloScanNHood`) or the index in the array of deltas (for `IloScanDeltas`) that corresponds to the solution delta leading to the current leaf node.

`public IlcNeighborIdentifierI * **getImpl**() const`

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

`public void **operator=**(const IlcNeighborIdentifier & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

`public void **setAtDelta**(IloSolution delta)`

This member function sets the stored delta to `delta`. Normally this action is performed by `IloScanNHood` or `IloScanDeltas`. As a user, you should not normally need to use this member function.

`public void **setAtIndex**(IloInt index)`

This member function sets the stored index to `index`. Normally this action is performed by `IloScanNHood` or `IloScanDeltas`. As a user, you should not normally need to use this member function.

Class IlcNodeEvaluator

Definition file: ilsolver/search.h
Include file: <ilsolver/ilosolver.h>

An instance of the class `IlcNodeEvaluator` represents a node evaluator, a function that evaluates open nodes before they are stored. A node evaluation is a value of type `IlcFloat`. A node evaluation comes into play after a failure when the solver asks for a new open node to explore. Solver selects the node with the minimum evaluation to be the new open node to explore.

See Also: IloSolver, IlcBFSEvaluator, IlcDDSEvaluator, IlcDFSEvaluator, IlcIDFSEvaluator, IlcSBSEvaluator

Constructor Summary	
`public`	`IlcNodeEvaluator()`
`public`	`IlcNodeEvaluator(IlcNodeEvaluatorI * impl)`

Method Summary	
`public IlcNodeEvaluatorI *`	`getImpl() const`
`public void`	`operator=(const IlcNodeEvaluator & h)`

Constructors

`public` **`IlcNodeEvaluator`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcNodeEvaluator`**`(IlcNodeEvaluatorI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

Methods

`public IlcNodeEvaluatorI *` **`getImpl`**`() const`

This constructor creates an object by copying another one.

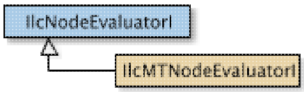
This member function returns a pointer to the implementation object of the invoking handle.

`public void` **`operator=`**`(const IlcNodeEvaluator & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IlcNodeEvaluatorI

Definition file: ilsolver/search.h
Include file: <ilsolver/ilosolver.h>



A node evaluator is an object in Solver. Like other Solver entities, a node evaluator is implemented by means of two classes: a handle class and an implementation class. In other words, an instance of the class `IlcNodeEvaluator` (a handle) contains a data member (the handle pointer) that points to an instance of the class `IlcNodeEvaluatorI` (its implementation object).

A node evaluator is linked to the life cycle of an open node. When a node is created, the method `evaluate` is called to evaluate the node. When the solver has to decide if it should jump to another node, the method `subsume` is called.

A node evaluator has its own life cycle:

- To be used internally by Solver code, it must be cloned by the member function `duplicate`.
- When the node evaluator is activated, Solver calls the member function `init`.

See Also: IlcNodeEvaluator, IlcSearchNode, IloApply

Constructor and Destructor Summary	
public	IlcNodeEvaluatorI(IloSolver solver, IlcBool duplicate)
public	~IlcNodeEvaluatorI()

Method Summary	
public virtual IlcNodeEvaluatorI *	duplicateEvaluator(IloSolver solver)
public virtual IlcFloat	evaluate(const IlcSearchNode n) const
public virtual void	init(const IlcSearchNode node)
public virtual IlcBool	subsume(IlcFloat evalBest, IlcFloat evalCurrent) const

Constructors and Destructors

`public` **`IlcNodeEvaluatorI`**`(IloSolver solver, IlcBool duplicate)`

This constructor creates an instance of the class `IlcNodeEvaluatorI` using `solver`. The parameter `duplicate` indicates whether the object is an internal copy used by the solver (`duplicate = IlcTrue`), or whether it is just a template. This constructor should not be called directly, as this class is an abstract class. This constructor is called automatically in the constructor of its subclasses.

`public` **`~IlcNodeEvaluatorI`**`()`

This destructor is called automatically by the destructor of its subclasses. It frees memory used by the invoking objects.

Methods

`public virtual IlcNodeEvaluatorI * duplicateEvaluator(IloSolver solver)`

This member function is called internally to duplicate the current node evaluator. When you use this member function, the `duplicate` parameter in the `IlcNodeEvaluatorI` constructor should be equal to `IlcTrue`.

`public virtual IlcFloat evaluate(const IlcSearchNode n) const`

When an open node `n` is created, this member function is called to evaluate the node. It returns a floating-point value which is the evaluation of the node. An evaluation of `IlcInfinity` means that the node should be discarded.

When you implement your own version of this virtual member function, you should make sure that your implementation is independent of the state of the solver (the instance of `IloSolver` where the invoking node evaluator is working). This signature includes `const` for that reason to prevent accumulated effects of multiple calls to this member function.

`public virtual void init(const IlcSearchNode node)`

When the goal `IlcApply` executes, it calls this method with the current node, `node`, being passed as parameter. The purpose of this method is to store a reference state for the node evaluator.

`public virtual IlcBool subsume(IlcFloat evalBest, IlcFloat evalCurrent) const`

During the search, the solver must decide if it should postpone the evaluation of the current node and move to another open node. This decision is based on the result of the `subsume` method. The `subsume` method is called with two floating-point values as parameters. The first value corresponds to the evaluation of the best open node, the second to the evaluation of the current node. This method returns `IlcTrue` to indicate that the solver should postpone the evaluation of the current node and continue with the evaluation of the best open node.

When you implement your own version of this member function, you should make sure that your implementation is independent of the state of the solver (the instance of `IloSolver` where the invoking node evaluator is working). Your implementation of subsume should decide statically whether one node subsumes another. This signature includes `const` for that reason to prevent accumulated effects of multiple calls to this member function.

Class IlcPathTransit

Definition file: ilsolver/ilcpath.h
Include file: <ilsolver/ilosolver.h>

Solver lets you define the *transit function* in a path constraint (the cost for linking two nodes together).

This class is the handle class of the object that defines this transit function.

An object of this handle class uses the virtual member function `IlcPathTransitI::transit` from its implementation class to define the transit function to apply in the path constraint in which it is used.

See Also: IlcPathLength, IlcPathTransitEvalI, IlcPathTransitFunction, IlcPathTransitI

Constructor Summary	
public	`IlcPathTransit(const IlcPathTransit & trans)`
public	`IlcPathTransit(IlcPathTransitI * trans)`
public	`IlcPathTransit(IloSolver s, IlcPathTransitFunction func)`

Method Summary	
public IlcPathTransitI *	`getImpl() const`

Constructors

`public` **`IlcPathTransit`**`(const IlcPathTransit & trans)`

This copy constructor accepts a reference to an implementation object and creates the corresponding handle object.

`public` **`IlcPathTransit`**`(IlcPathTransitI * trans)`

This constructor creates a handle that corresponds to the same implementation object that `trans` indicates.

`public` **`IlcPathTransit`**`(IloSolver s, IlcPathTransitFunction func)`

This constructor creates a new transit function from an evaluation function. The implementation object of the newly created handle is an instance of the class `IlcPathTransitEvalI` constructed with the evaluation function indicated by the argument `func`.

Methods

`public IlcPathTransitI *` **`getImpl`**`() const`

This member function returns a pointer to the implementation object corresponding to the invoking handle.

Example

The simplest way to define a new transit function is to define an evaluation function. For example, the following code defines a transit function that returns the distance from node `i` to node `j`.

```
IlcFloat** Distance;
IlcFloat GetDistance(IlcInt i, IlcInt j){
    return Distance[i][j];
}
```

Then you create a transition object like this:

```
IlcPathTransit transition(GetDistance);
```

However, this approach is not very efficient if the transit function depends on more than the indexes of two nodes. In that example, you saw that we needed to have a global distance array, a situation that we generally want to avoid. Another approach is to define a new transit function like this: define the virtual function `transit` for this subclass, as well as a function that returns a handle.

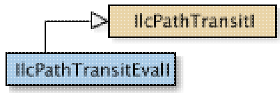
```
#include <ilsolver/ilcpath.h>
class IlcGetDistanceI: public IlcPathTransitI {
    IlcFloat** distance;
  public:
    IlcGetDistanceI(); //Allocates and fills distance
    virtual IlcFloat transit(IlcInt i, IlcInt j);
};
IlcFloat transit(IlcInt i, IlcInt j) {
    return distance[i][j];
}
IlcPathTransit GetDistance(IloSolver s) {
    return new (s.getHeap()) IlcGetDistanceI();
}
```

Then with the newly defined class, you create a transition object like this:

```
IlcPathTransit distance = GetDistance(s);
```

Class IlcPathTransitEvalI

Definition file: ilsolver/ilcpath.h
Include file: <ilsolver/ilosolver.h>



Solver lets you define the *transit function* in a path constraint (the cost for linking two nodes together).

This class is an implementation class, a predefined subclass of `IlcPathTransitI`, that you use to define a new transition function expressed by an evaluation function. This evaluation function is of type `IlcPathTransitFunction`.

See Also: IlcPathLength, IlcPathTransit, IlcPathTransitFunction, IlcPathTransitI

Constructor Summary	
`public`	`IlcPathTransitEvalI(IlcPathTransitFunction function)`

Method Summary	
`public virtual IlcFloat`	`transit(IlcInt i, IlcInt j)`

Inherited Methods from `IlcPathTransitI`
`transit`

Constructors

`public **IlcPathTransitEvalI**(IlcPathTransitFunction function)`

This constructor creates a new transit function from an evaluation function. Objects of this class use the evaluation function indicated by the argument `function` to define the transition costs used in the path constraint.

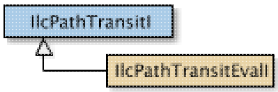
Methods

`public virtual IlcFloat **transit**(IlcInt i, IlcInt j)`

This virtual member function returns the transition cost from node `i` to node `j`. In order to do this, it calls the evaluation function.

Class IlcPathTransitI

Definition file: ilsolver/ilcpath.h
Include file: <ilsolver/ilosolver.h>



Solver lets you define the transit function in a path constraint (the cost for linking two nodes together).

This class is the implementation class for `IlcPathTransit`, the class of object that defines a transit function for the path constraint. The virtual member function in `IlcPathTransitI` returns the transition cost for connecting two nodes together.

To express new transit functions, you can define a subclass of `IlcPathTransitI`. If this transition can be expressed by an evaluation function, then you can use the predefined subclass `IlcPathTransitEvalI` for that purpose.

See Also: IlcPathLength, IlcPathTransit, IlcPathTransitEvalI, IlcPathTransitFunction

Constructor and Destructor Summary	
`public`	`IlcPathTransitI()`
`public`	`~IlcPathTransitI()`

Method Summary	
`public virtual IlcFloat`	`transit(IlcInt i, IlcInt j)`

Constructors and Destructors

`public` **`IlcPathTransitI`**`()`

This constructor creates an implementation object.

`public` **`~IlcPathTransitI`**`()`

As this class is to be subclassed, a virtual destructor is provided

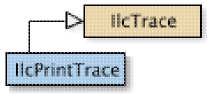
Methods

`public virtual IlcFloat` **`transit`**`(IlcInt i, IlcInt j)`

This virtual member function returns the transition cost from node `i` to node `j`. Its default implementation returns 0 (zero) as the value of every transition.

Class IlcPrintTrace

Definition file: ilsolver/ilctrace.h
Include file: <ilsolver/ilctrace.h>



An instance of this class is part of the Solver trace mechanism. An instance of this class prints trace events.

Example

Here is an example of how to use a print trace.

```
IlcPrintTrace trace(solver, IlcTraceVars | IlcTraceFail);
```

See Also: IlcTrace, IlcTraceI

Constructor and Destructor Summary	
public	IlcPrintTrace(IloSolver s, IlcUInt flags=4, const char * name=0)
public	IlcPrintTrace(IlcPrintTraceI * impl)

Method Summary	
public IlcPrintTraceI *	getImpl() const
public void	operator=(const IlcPrintTrace & trace)
public const IlcPrintTrace &	trace(IlcAnySetVar) const
public const IlcPrintTrace &	trace(IlcIntSetVar) const
public const IlcPrintTrace &	trace(IlcFloatVar) const
public const IlcPrintTrace &	trace(IlcAnyVar) const
public const IlcPrintTrace &	trace(IlcIntVar) const

Constructors and Destructors

`public IlcPrintTrace(IloSolver s, IlcUInt flags=4, const char * name=0)`

This constructor creates a print trace. The parameter `flags` indicates which events it traces. See the topic Trace Events in the class `IlcTraceI` for a list of the possible flags.

`public IlcPrintTrace(IlcPrintTraceI * impl)`

This constructor creates a handle (an instance of the class `IlcPrintTrace`) from a pointer to an implementation object (an instance of the implementation class `IlcPrintTraceI`).

Methods

`public IlcPrintTraceI * getImpl() const`

This member function returns a pointer to the implementation object of the invoking print trace.

`public void` **`operator=`**`(const IlcPrintTrace & trace)`

This operator assigns `trace` to the invoking print trace. After calling this operator, both handles point to the same implementation object.

`public const IlcPrintTrace &` **`trace`**`(IlcAnySetVar) const`

These member functions hook a variable; that is, they offer a link between a print trace and a variable. (For users of previous versions of Solver, this member function resembles the trace hook mechanism.)

`public const IlcPrintTrace &` **`trace`**`(IlcIntSetVar) const`

These member functions hook a variable; that is, they offer a link between a print trace and a variable. (For users of previous versions of Solver, this member function resembles the trace hook mechanism.)

`public const IlcPrintTrace &` **`trace`**`(IlcFloatVar) const`

These member functions hook a variable; that is, they offer a link between a print trace and a variable. (For users of previous versions of Solver, this member function resembles the trace hook mechanism.)

`public const IlcPrintTrace &` **`trace`**`(IlcAnyVar) const`

These member functions hook a variable; that is, they offer a link between a print trace and a variable. (For users of previous versions of Solver, this member function resembles the trace hook mechanism.)

`public const IlcPrintTrace &` **`trace`**`(IlcIntVar) const`

These member functions hook a variable; that is, they offer a link between a print trace and a variable. (For users of previous versions of Solver, this member function resembles the trace hook mechanism.)

Class IlcRandom

Definition file: ilsolver/random.h
Include file: ilsolver/random.h

Objects of this class produce streams of pseudo-random numbers. You can use objects of this class to create a search with a random element. This class will produce the same stream of random numbers in recompute mode as when it was in compute mode.

For more information, see the member function `IloSolver::isInRecomputeMode`.

For more information, see `IloRandom`.

See Also: IloRandom

Constructor Summary	
`public`	`IlcRandom()`
`public`	`IlcRandom(IlcRandomI * impl)`
`public`	`IlcRandom(IloSolver m, IlcInt seed=0)`

Method Summary	
`public void`	`copy(IloRandom rnd)`
`public void`	`copy(IlcRandom rnd)`
`public void`	`copyTo(IloRandom rnd)`
`public IlcFloat`	`getFloat() const`
`public IlcRandomI *`	`getImpl() const`
`public IlcInt`	`getInt(IlcInt n) const`
`public const char *`	`getName() const`
`public IlcAny`	`getObject() const`
`public IloSolver`	`getSolver() const`
`public IloSolverI *`	`getSolverI() const`
`public void`	`operator=(const IlcRandom & h)`
`public void`	`reSeed(IlcInt seed)`
`public void`	`setName(const char * name) const`
`public void`	`setObject(IlcAny object) const`

Constructors

`public` **`IlcRandom`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcRandom`**`(IlcRandomI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public IlcRandom(IloSolver m, IlcInt seed=0)`

This constructor creates a random number generator, initially seeded with the seed `seed`.

Methods

`public void copy(IloRandom rnd)`

This member function copies the state of a Concert Technology random number generator to the invoking one. After the copy, both generators will produce the same stream of pseudo-random numbers.

`public void copy(IlcRandom rnd)`

This member function copies the state of another random number generator to the invoking one. After the copy, both generators will produce the same stream of pseudo-random numbers.

`public void copyTo(IloRandom rnd)`

This member function copies the state of the invoking generator to a Concert Technology random number generator. After the copy, both generators will produce the same stream of pseudo-random numbers.

`public IlcFloat getFloat() const`

This member function returns a floating point number drawn uniformly from the range `[0..1)`.

`public IlcRandomI * getImpl() const`

This constructor creates an object by copying another one. This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

`public IlcInt getInt(IlcInt n) const`

This member function returns a integer number drawn uniformly from the range `[0..n)`.

`public const char * getName() const`

This member function returns the name of the invoking object.

`public IlcAny getObject() const`

This member function returns a pointer to the external object associated with the invoking object, if there is such an association. It returns 0 (zero) otherwise.

`public IloSolver getSolver() const`

This member function returns an instance of `IloSolver` associated with the invoking object.

`public IloSolverI * getSolverI() const`

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

`public void **operator=**(const IlcRandom & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

`public void **reSeed**(IlcInt seed)`

This member function reseeds the generator with seed `seed`.

`public void **setName**(const char * name) const`

This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

`public void **setObject**(IlcAny object) const`

This member function establishes a link between the invoking object and an external object of which the invoking object might be a data member.

Class IlcRevAny

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>

`IlcRevAny` is a reversible version of the basic predefined Solver type, `IlcAny`. The reversible version makes it easier to restore a previous state. This class has a value data member corresponding to `IlcAny`. This value data member is automatically restored when Solver backtracks.

This class is *not* a handle class. Objects of this class should be used directly, that is, not through pointers to them. Do *not* create instances of `IlcRevAny` as automatic objects (that is, as objects allocated on the C++ stack). Do *not* pass them by value.

An object of the class `IlcRevAny` is automatically cast to the basic type `IlcAny`, if needed. Instances of this reversible class can be used as data members. However, *do not use them as types for automatic variables*, where “automatic” has its usual C++ meaning, as this practice would create memory-access errors. Do *not* pass them as arguments.

An instance of `IlcRevAny` behaves very much like an instance of `IlcAny`; the difference in their behavior has to do with assignments. Indeed, all reversible assignments are undone when Solver backtracks.

Unfortunately, the class `IlcRevAny` is not typed. The pointers are of type `IlcAny` and thus give no indication of the type of objects they point to. See the macro `ILCREV` for a way to define a reversible class for typed pointers yourself.

For more information, see the concepts State and Reversibility.

See Also: IlcRevBool, IlcRevFloat, IlcRevInt

Constructor Summary	
`public`	`IlcRevAny(IloSolver s, IlcAny initValue=0)`

Method Summary	
`public IlcAny`	`getValue() const`
`public`	`operator IlcAny() const`
`public void`	`setValue(IloSolver solver, IlcAny value)`

Constructors

`public` **`IlcRevAny`**`(IloSolver s, IlcAny initValue=0)`

The constructor creates a new object, an instance of `IlcRevAny`. It is more memory-efficient than the constructor without arguments.

Methods

`public IlcAny` **`getValue`**`() const`

This member function accesses the value of the invoking instance of `IlcRevAny`.

```
public operator IlcAny() const
```

This operator returns the value of the instance of `IlcRevAny`. In other words, this operator automatically casts an instance of `IlcRevAny` into an instance of `IlcAny`.

```
public void setValue(IloSolver solver, IlcAny value)
```

This member function modifies the value of the invoking object by reversibly assigning `value` to it. When Solver backtracks, this reversible modification will be undone.

Class IlcRevBool

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>

`IlcRevBool` is a reversible version of the basic predefined Solver type, `IlcBool`. The reversible version makes it easier to restore a previous state. This class has a value data member corresponding to `IlcBool`. This value data member is automatically restored when Solver backtracks.

This class is *not* a handle class. Objects of this class should be used directly, that is, not through pointers to them. Do *not* create instances of `IlcRevBool` as automatic objects (that is, as objects allocated on the C++ stack). Do *not* pass them by value.

An object of the class `IlcRevBool` is automatically cast to the basic type `IlcBool`, if needed. Instances of this reversible class can be used as data members. However, *do not use them as types for automatic variables*, where "automatic" has its usual C++ meaning, as this practice would create memory-access errors. Do *not* pass them as arguments.

An instance of `IlcRevBool` behaves very much like an instance of `IlcBool`; the difference in their behavior has to do with assignments. Indeed, all reversible assignments are undone when Solver backtracks.

For more information, see the concepts State and Reversibility.

See Also: IlcRevAny, IlcRevFloat, IlcRevInt

Constructor Summary	
public	`IlcRevBool(IloSolver solver, IlcBool initValue=IlcFalse)`

Method Summary	
public IlcBool	`getValue() const`
public	`operator IlcBool() const`
public void	`setValue(IloSolver solver, IlcBool value)`

Constructors

`public` **`IlcRevBool`**`(IloSolver solver, IlcBool initValue=IlcFalse)`

The constructor creates a new object, an instance of `IlcRevBool`. It is more memory-efficient than the constructor without arguments.

Methods

`public IlcBool` **`getValue`**`() const`

This member function accesses the value of the instance of `IlcRevBool`.

`public` **`operator IlcBool`**`() const`

This operator returns the value of the instance of `IlcRevBool`. In other words, this operator automatically casts an instance of `IlcRevBool` into an instance of `IlcBool`.

`public void` **`setValue`**`(IloSolver solver, IlcBool value)`

This operator modifies the value of an instance of `IlcRevBool` by assigning `value` to it. When Solver backtracks, this reversible modification will be undone.

Class IlcRevFloat

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>

`IlcRevFloat` is a reversible version of the basic predefined Solver type, `IlcFloat`. The reversible version makes it easier to restore a previous state. This class has a value data member corresponding to `IlcFloat`. This value data member is automatically restored when Solver backtracks.

This class is *not* a handle class. Objects of this class should be used directly, that is, not through pointers to them. Do *not* create instances of `IlcRevFloat` as automatic objects (that is, as objects allocated on the C++ stack). Do *not* pass them by value.

An object of the class `IlcRevFloat` is automatically cast to the basic type `IlcFloat`, if needed. Instances of this reversible class can be used as data members. However, *do not use them as types for automatic variables*, where "automatic" has its usual C++ meaning, as this practice would create memory-access errors. Do *not* pass them as arguments.

An instance of `IlcRevFloat` behaves very much like an instance of `IlcFloat`; the difference in their behavior has to do with assignments. Indeed, all reversible assignments are undone when Solver backtracks.

For more information, see the concepts State and Reversibility.

See Also: IlcRevAny, IlcRevBool, IlcRevInt

Constructor Summary	
`public`	`IlcRevFloat()`
`public`	`IlcRevFloat(IloSolver solver, IlcFloat initValue=0.)`

Method Summary	
`public IlcFloat`	`getValue() const`
`public`	`operator IlcFloat() const`
`public void`	`setValue(IloSolver solver, IlcFloat value)`

Constructors

`public` **`IlcRevFloat`**`()`

The constructor creates a new object, an instance of `IlcRevFloat`. It is less memory-efficient than the constructor with a solver as its argument.

`public` **`IlcRevFloat`**`(IloSolver solver, IlcFloat initValue=0.)`

The constructor creates a new object, an instance of `IlcRevFloat`. It is more memory-efficient than the constructor without arguments.

Methods

`public IlcFloat getValue() const`

This member function accesses the value of the instance of `IlcRevFloat`.

`public operator IlcFloat() const`

This operator returns the value of the instance of `IlcRevFloat`. In other words, this operator automatically casts an instance of `IlcRevFloat` into an instance of `IlcFloat`.

`public void setValue(IloSolver solver, IlcFloat value)`

This member function modifies the value of the invoking object by reversibly assigning `value` to it. When Solver backtracks, this reversible modification will be undone.

Class IlcRevInt

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>

`IlcRevInt` is a reversible version of the basic predefined Solver type, `IlcInt`. The reversible version makes it easier to restore a previous state. This class has a value data member corresponding to `IlcInt`. This value data member is automatically restored when Solver backtracks.

This class is *not* a handle class. Objects of this class should be used directly, that is, not through pointers to them. Do *not* create instances of `IlcRevInt` as automatic objects (that is, as objects allocated on the C++ stack). Do *not* pass them by value.

An object of the class `IlcRevInt` is automatically cast to the basic type `IlcInt`, if needed. Instances of this reversible class can be used as data members. However, *do not use them as types for automatic variables*, where "automatic" has its usual C++ meaning, as this practice would create memory-access errors. Do *not* pass them as arguments.

An instance of `IlcRevInt` behaves very much like an instance of `IlcInt`; the difference in their behavior has to do with assignments. Indeed, all reversible assignments are undone when Solver backtracks.

For more information, see the concepts State and Reversibility.

See Also: IlcRevAny, IlcRevBool, IlcRevFloat

Constructor Summary	
`public`	`IlcRevInt(IloSolver solver, IlcInt initValue=0)`

Method Summary	
`public IlcInt`	`getValue() const`
`public`	`operator IlcInt() const`
`public void`	`setValue(IloSolver solver, IlcInt value)`

Constructors

`public` **`IlcRevInt`**`(IloSolver solver, IlcInt initValue=0)`

The constructor creates a new object, an instance of `IlcRevInt`. It is more memory-efficient than the constructor without arguments.

Methods

`public IlcInt` **`getValue`**`() const`

This member function accesses the value of the instance of `IlcRevInt`.

`public` **`operator IlcInt`**`() const`

This operator returns the value of the instance of `IlcRevInt`. In other words, this operator automatically casts an instance of `IlcRevInt` into an instance of `IlcInt`.

```
public void setValue(IloSolver solver, IlcInt value)
```

This member function modifies the value of the invoking object by reversibly assigning `value` to it. When Solver backtracks, this reversible modification will be undone.

Class IlcSearchLimit

Definition file: ilsolver/search.h
Include file: <ilsolver/ilosolver.h>

The class `IloSearchLimit` represents search limits in a Concert Technology model. The class `IlcSearchLimit` represents search limits internally in a Solver search.

For more information, see the concept Global Modifiers.

See Also: IloSearchLimit

Constructor Summary	
`public`	`IlcSearchLimit()`
`public`	`IlcSearchLimit(IlcSearchLimitI * impl)`

Method Summary	
`public IlcSearchLimitI *`	`getImpl() const`
`public void`	`operator=(const IlcSearchLimit & h)`

Constructors

`public` **`IlcSearchLimit`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcSearchLimit`**`(IlcSearchLimitI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

Methods

`public IlcSearchLimitI *` **`getImpl`**`() const`

This constructor creates an object by copying another one.

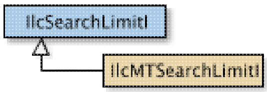
This member function returns a pointer to the implementation object of the invoking handle.

`public void` **`operator=`**`(const IlcSearchLimit & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IlcSearchLimitI

Definition file: ilsolver/search.h
Include file: <ilsolver/ilosolver.h>



A search limit is an object in Solver. Like other Solver entities, a search limit is implemented by means of two classes: a handle class and an implementation class. In other words, an instance of the class `IlcSearchLimit` (a handle) contains a data member (the handle pointer) that points to an instance of the class `IlcSearchLimitI` (its implementation object).

A search limit is used to prune part of the search tree. The member function `check` indicates whether the limit has been reached.

A search limit has its own life cycle:

- To be used internally, it has to be cloned by the member function `duplicate`.
- When it is activated, the member function `init` is called.

The class `IlcSearchLimitI` is extendable; in other words, you can write your own search limits. For an example of how to do so, see the *IBM ILOG Solver User's Manual*.

See Also: IlcSearchLimit, IlcSearchNode, IloLimitSearch, IloSearchLimitI

Constructor and Destructor Summary	
public	IlcSearchLimitI(IloSolver solver, IlcBool duplicate)
public	~IlcSearchLimitI()

Method Summary	
public virtual IlcBool	check(const IlcSearchNode node) const
public virtual IlcSearchLimitI *	duplicateLimit(IloSolver solver)
public virtual void	init(const IlcSearchNode node)

Constructors and Destructors

`public` **`IlcSearchLimitI`**`(IloSolver solver, IlcBool duplicate)`

This constructor creates an instance of the class `IlcSearchLimitI` using `solver`. The parameter `duplicate` indicates if the object is an internal copy used by the solver (`duplicate = IlcTrue`), or if it is just a template. This constructor should not be called directly as this class is an abstract class. This constructor is called automatically in the constructor of its subclasses.

`public` **`~IlcSearchLimitI`**`()`

This destructor is called automatically by the destructor of its subclasses. It frees memory used by the objects.

Methods

```
public virtual IlcBool check(const IlcSearchNode node) const
```

This member function is called with the current node as parameter to check whether the limit implemented by the object has been reached. If the limit has been reached, this virtual member function should return `IlcTrue`. Afterwards, the unexplored search tree covered by this limit will be simply discarded.

When you implement this virtual member function yourself, you should make sure that your implementation decides exactly one time (that is, once and for all) whether a limit has been reached. Once the limit has been reached, it will not be called again, and all nodes covered by the limit will be discarded. The signature includes `const` for that reason to avoid accumulated effects of multiple calls of this member function.

```
public virtual IlcSearchLimitI * duplicateLimit(IloSolver solver)
```

This member function is called internally to duplicate the current search limit. When you use this function, the `duplicate` parameter in the `IlcSearchLimitI` constructor should be equal to `IlcTrue`.

```
public virtual void init(const IlcSearchNode node)
```

When the goal `IlcLimitSearch` executes, it calls this method with the current `node` passed as its parameter. The purpose of this method is to store a reference state for the node evaluator.

Class IlcSearchMonitor

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>

Instances of this class represent search monitors. A search monitor watches and reports events in a search. See the virtual member functions of its implementation class, `IlcSearchMonitorI`, to get an idea of what monitors can do in your application.

See Also: IlcSearchMonitorI

Constructor Summary	
public	`IlcSearchMonitor()`
public	`IlcSearchMonitor(IlcSearchMonitorI * impl)`

Method Summary	
public IlcSearchMonitorI *	`getImpl() const`
public void	`operator=(const IlcSearchMonitor & h)`

Constructors

public **IlcSearchMonitor**()

This constructor creates a handle (an instance of the class `IlcSearchMonitor`) from a pointer to an implementation object (an instance of the implementation class `IlcSearchMonitorI`).

This constructor creates an empty handle. You must initialize it before you use it.

public **IlcSearchMonitor**(IlcSearchMonitorI * impl)

This constructor creates a handle object from a pointer to an implementation object.

Methods

public IlcSearchMonitorI * **getImpl**() const

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

public void **operator=**(const IlcSearchMonitor & h)

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IlcSearchMonitorI

Definition file: ilsolver/search.h
Include file: <ilsolver/ilosolver.h>

A search monitor watches and reports events in a search. Specifically, it offers member functions capable of reporting such events as the creation of a search node, the destruction of a search node, failure of the search, and so forth.

A search monitor is an object in Solver. Like other Solver entities, a search monitor is implemented by means of two classes: a handle class and an implementation class. In other words, an instance of the class `IlcSearchMonitor` (a handle) contains a data member (the handle pointer) that points to an instance of the class `IlcSearchMonitorI` (its implementation object).

See Also: IlcNodeEvaluator, IlcSearchLimit, IlcSearchMonitor, IlcSearchNode, IlcSearchSelector, IlcSearchSelectorI, IloApply, IloLimitSearch, IloSelectSearch

Constructor and Destructor Summary	
`public`	`IlcSearchMonitorI()`
`public`	`~IlcSearchMonitorI()`

Method Summary	
`protected virtual void`	`whenBeginMove(const IlcSearchNode node)`
`protected virtual void`	`whenCreate(const IlcSearchNode node)`
`protected virtual void`	`whenEndMove(const IlcSearchNode node)`
`protected virtual void`	`whenFail(const IlcSearchNode node)`
`protected virtual void`	`whenInMove(const IlcSearchNode node)`
`protected virtual void`	`whenOpen(const IlcSearchNode node, IlcBool recompute)`
`protected virtual void`	`whenPrune(const IlcSearchNode node, IlcSearchMonitorI::IlcPruneMode mode)`
`protected virtual void`	`whenSolution(const IlcSearchNode node)`

Inner Enumeration
IlcSearchMonitorI::IlcPruneMode

Constructors and Destructors

`public` **`IlcSearchMonitorI`**`()`

This constructor creates an instance of the class `IlcSearchMonitorI`. This constructor should not be called directly as this class is an abstract class. This constructor is called automatically in the constructor of its subclasses.

`public` **`~IlcSearchMonitorI`**`()`

As this class is to be subclassed, a virtual destructor is provided.

Methods

`protected virtual void` **`whenCreate`**`(const IlcSearchNode node)`

Solver calls this member function when it creates a node.

`protected virtual void` **`whenFail`**`(const IlcSearchNode node)`

Solver calls this member function when the search fails. Unless the search is already complete, Solver normally backtracks and recomputes after this member function.

`protected virtual void` **`whenSolution`**`(const IlcSearchNode node)`

Solver calls this member function when it exits successfully from the search.

`protected virtual void` **`whenPrune`**`(const IlcSearchNode node, IlcSearchMonitorI::IlcPruneMode mode)`

Solver calls this member function when it prunes the frontier of nodes in the search tree. The parameter `mode` indicates the reason for pruning when it prunes an open node belonging to the frontier of nodes in the search tree.

`protected virtual void` **`whenOpen`**`(const IlcSearchNode node, IlcBool recompute)`

Solver calls this member function when it explores a node. The parameter `recompute` indicates whether Solver is recomputing (`IlcTrue`) or not (`IlcFalse`) at the time of the call.

`protected virtual void` **`whenBeginMove`**`(const IlcSearchNode node)`

Solver calls this member function when it has decided to move from one node to in the search tree.

`protected virtual void` **`whenInMove`**`(const IlcSearchNode node)`

Solver calls this member function after it backtracks. The backtrack may be because of a failure or because of a move from one node to another.

`protected virtual void` **`whenEndMove`**`(const IlcSearchNode node)`

Solver calls this member function after a recomputation following a backtrack.

Inner Enumerations

Enumeration IlcPruneMode

Definition file: ilsolver/search.h
Include file: <ilsolver/ilosolver.h>

The values in this enumeration indicate why Solver prunes the nodes of a search tree.

See Also: IlcSearchMonitorI

Fields:

`searchNotFailed`

`searchFailedNormally`

`killedBySelector`

`killedByLimit`

`killedByLabel`

`killedByEvaluator`

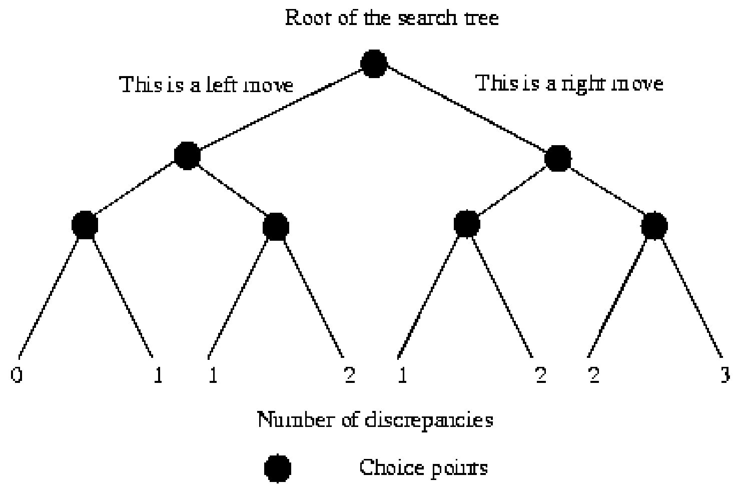
`killedByExit`

Class IlcSearchNode

Definition file: ilsolver/search.h
Include file: <ilsolver/ilosolver.h>

Instances of this class represent nodes in the search tree of a parallel search. Most of its member functions access information of use to *node evaluators*. Node evaluators are documented in the class `IlcNodeEvaluator`.

The following figure illustrates a search tree consisting of its root node and choice points. Its maximum *or-depth* is three. It shows the number of discrepancies at the leaves. A *discrepancy* corresponds to the or-depth of that search node minus its left-depth.



See Also: IlcNodeEvaluator

Constructor Summary	
public	`IlcSearchNode()`
public	`IlcSearchNode(IlcSearchNodeI * impl)`

Method Summary	
public IlcFloat	`getBitsetValue(IlcInt start, IlcInt length) const`
public IlcInt	`getDepth() const`
public IlcSearchNodeI *	`getImpl() const`
public IlcInt	`getLastDiscrepancyDepth(IlcInt offset) const`
public IlcInt	`getLeftDepth() const`
public IlcInt	`getRightDepth() const`
public IloSolver	`getSolver() const`
public IloSolverI *	`getSolverI() const`
public void	`operator=(const IlcSearchNode & h)`

Constructors

```
public IlcSearchNode()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IlcSearchNode(IlcSearchNodeI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

Methods

```
public IlcFloat getBitsetValue(IlcInt start, IlcInt length) const
```

Node evaluators use this member function. It first converts the path from the root to the node into a bitset, where `0` (zero) is a left move and `1` (one) a right move. Then it extracts from this bitset a sub-bitset starting from the position indicated by the parameter `start` and extending. It then converts this extracted bitset into an integer, using the convention that the least significant bit is to the left (corresponding to the choice points closer to the root of the search tree). This member function then casts this integer into a `IlcFloat` value and returns it.

```
public IlcInt getDepth() const
```

Node evaluators use this member function. It returns the or-depth of the invoking search node.

```
public IlcSearchNodeI * getImpl() const
```

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

```
public IlcInt getLastDiscrepancyDepth(IlcInt offset) const
```

Node evaluators use this member function. It returns the or-depth of a particular node in the search tree; that node is the one located at `offset` number of right moves along the path from the root of the search tree to the invoking node.

```
public IlcInt getLeftDepth() const
```

Node evaluators use this member function. It returns the number of left moves in the path starting from the root of the search tree to the position of the invoking node.

```
public IlcInt getRightDepth() const
```

Node evaluators use this member function. It returns the number of right moves in the path starting from the root of the search tree to the position of the invoking node.

`public IloSolver` **`getSolver`**`() const`

This member function returns the solver that the invoking node belongs to.

`public IloSolverI *` **`getSolverI`**`() const`

This member function returns the implementation of the solver that the invoking node belongs to.

`public void` **`operator=`**`(const IlcSearchNode & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IlcSearchSelector

Definition file: ilsolver/search.h
Include file: <ilsolver/ilosolver.h>

Instances of this class represent search selectors. A search selector acts as a filter during the search. The function `IloMinimizeVar`, for example, creates and returns a search selector for use in your applications.

See Also: IlcNodeEvaluator, IloFirstSolution, IloMinimizeVar, IloSelectSearch

Constructor Summary	
`public`	`IlcSearchSelector()`
`public`	`IlcSearchSelector(IlcSearchSelectorI * impl)`

Method Summary	
`public IlcSearchSelectorI *`	`getImpl() const`
`public void`	`operator=(const IlcSearchSelector & h)`

Constructors

`public` **`IlcSearchSelector`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcSearchSelector`**`(IlcSearchSelectorI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

Methods

`public IlcSearchSelectorI *` **`getImpl`**`() const`

This constructor creates an object by copying another one.

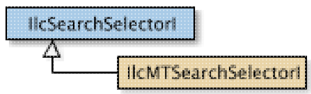
This member function returns a pointer to the implementation object of the invoking handle.

`public void` **`operator=`**`(const IlcSearchSelector & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IlcSearchSelectorI

Definition file: ilsolver/search.h
Include file: <ilsolver/ilosolver.h>



The class `IlcSearchSelector` represents search selectors internally in the Solver search. The class `IloSearchSelector` represents search selectors in a Concert Technology model. Search selectors are useful as filters during a Solver search.

A search selector is an object in Solver. Like other Solver entities, a search selector is implemented by means of two classes: a handle class and an implementation class. In other words, an instance of the class `IlcSearchSelector` (a handle) contains a data member (the handle pointer) that points to an instance of the class `IlcSearchSelectorI` (its implementation object).

A search selector has several purposes:

- To implement a minimization process using the member functions `update`, `updateTo`, and `saveObjectiveValue`.
- To implement a filter on open nodes using the member functions `evaluate` and `isFeasible`.
- To manage selected nodes using the member functions `registerSolution`, `whenFinished`, `getCurrentNode`, `activateNode`, and `closeNode`.

A search selector has its own life cycle. To be used internally, it has to be cloned by the `duplicate` member function.

See Also: IlcSearchNode, IlcSearchSelector

Constructor and Destructor Summary	
`public`	`IlcSearchSelectorI(IloSolver solver, IlcBool duplicate)`
`public`	`~IlcSearchSelectorI()`

Method Summary	
`public void`	`activateNode(IloSolver solver, IlcSearchNode n)`
`public void`	`closeNode(IlcSearchNode n)`
`public virtual IlcSearchSelectorI *`	`duplicateSelector(IloSolver solver)`
`public virtual IlcFloat`	`evaluate(const IlcSearchNode node) const`
`public IlcSearchNode`	`getCurrentNode(IloSolver solver)`
`public virtual IlcBool`	`isFeasible(IlcFloat eval) const`
`public void`	`saveObjectiveValue(IloSolver solver, IloNum newVal)`
`public virtual void`	`updateObjective(IloSolver solver)`
`public virtual void`	`updateObjectiveTo(IloSolver solver, IloNum newVal)`
`public virtual void`	`whenFinishedTree(IloSolver solver, IlcBool toKill)`
`public virtual void`	`whenLeaf(IloSolver solver)`

Constructors and Destructors

`public IlcSearchSelectorI(IloSolver solver, IlcBool duplicate)`

This constructor creates an instance of the class `IlcSearchSelectorI` using the `solver`. The parameter `duplicate` indicates whether the object is an internal copy used by the `solver` (`duplicate = IlcTrue`), or if it is just a template. This constructor should not be called directly as this class is an abstract class. This constructor is called automatically in the constructor of its subclasses.

`public ~IlcSearchSelectorI()`

This destructor is called automatically by the destructor of its subclasses. It frees memory used by the objects.

Methods

`public void activateNode(IloSolver solver, IlcSearchNode n)`

This member function should be called only from within the `whenFinished` member function. It indicates that this node `n` is in fact selected (which is the purpose of the `IlcSelectSearch` function).

`public void closeNode(IlcSearchNode n)`

This member function deletes a node `n` obtained with the getCurrentNode member function.

`public virtual IlcSearchSelectorI * duplicateSelector(IloSolver solver)`

This member function is called internally to duplicate the current search selector. When you use this member function, the `duplicate` parameter in the `IlcSearchSelectorI` constructor should be equal to `IlcTrue`.

`public virtual IlcFloat evaluate(const IlcSearchNode node) const`

This member function evaluates the node given as a parameter. This evaluation will be used by the `isFeasible` member function. Its purpose is to prune nodes before the solver jumps to them.

When you implement this virtual member function yourself, you should make sure that your implementation is independent of the state of the solver (the instance of `IloSolver` where the invoking search selector is working). The signature of this member function includes `const` for that reason to avoid accumulated effects of multiple calls of this member function.

`public IlcSearchNode getCurrentNode(IloSolver solver)`

This member function returns a copy of the current search node. It should be called only from the `registerSolution` member function. To discard a node obtained by this member function, you must use the `closeNode` member function.

`public virtual IlcBool isFeasible(IlcFloat eval) const`

This member function uses the `eval` parameter which comes from the `evaluate` member function. A return value of `IlcFalse` indicates that it is safe to discard this node and not to evaluate it. The return value `IlcTrue` indicates that the node will be kept.

When you implement this virtual member function yourself, you should make sure that your implementation is independent of the state of the solver (the instance of `IloSolver` where the invoking search selector is working). The signature of this member function includes `const` for that reason to avoid accumulated effects of multiple calls of this member function.

`public void` **`saveObjectiveValue`**`(IloSolver solver, IloNum newVal)`

This member function is called by the solver when it arrives at a leaf of the search tree associated with the goal given as a parameter to the `IlcSelectSearch` function. It may use the member functions `getCurrentNode` and `closeNode`.

`public virtual void` **`updateObjective`**`(IloSolver solver)`

This member function is used to implement a minimization process. It checks whether a better upper bound on the objective is known. If this is the case, it adds the corresponding constraint and saves the information using the `saveObjectiveValue` member function.

`public virtual void` **`updateObjectiveTo`**`(IloSolver solver, IloNum newVal)`

This member function is called during recomputation. It adds the same constraint as was posted by the `update` member function. The parameter `newVal` is the one which was saved by the `saveObjectiveValue` member function.

`public virtual void` **`whenFinishedTree`**`(IloSolver solver, IlcBool toKill)`

This member function is called by the `solver` when it has completely explored the search tree associated with the goal given as a parameter to the `IlcSelectSearch` function. It may use the member functions `closeNode` and `activateNode`.

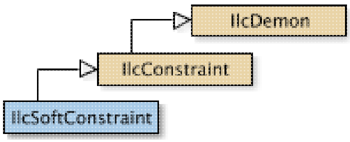
The expected behavior depends on the parameter `toKill`. If `toKill` is equal to `IlcFalse` (which is the usual case), the search selector proceeds normally. If `toKill` is equal to `IlcTrue`, the search selector must not activate any of the stored nodes, but must kill them all using the `closeNode` member function.

`public virtual void` **`whenLeaf`**`(IloSolver solver)`

This member function is called by the `solver` when it arrives at a leaf of the search tree associated with the goal given as a parameter to the `IlcSelectSearch` function. It may use the member functions `getCurrentNode` and `closeNode`.

Class IlcSoftConstraint

Definition file: ilsolver/disjunct.h



An instance of the class `IlcSoftConstraint` can only be created using the member function `IlcSoftConstraint IlcSoftCtHandler::createSoftConstraint(IlcConstraintI*).`

The class `IlcSoftCtHandler` is used to manage the definition of a soft constraint. It contains information about the copied constraint, the copied variables and their relation to the original constraint and the original variables.

Method Summary	
`public IlcInt`	`getCopiedVarInProcess() const`
`public IlcSoftConstraintI *`	`getImpl() const`
`public IlcIntVar`	`getStatusVar()`
`public IlcBool`	`isFailed()`
`public void`	`whenDomainReduction(IlcDemon demon)`
`public void`	`whenFail(IlcDemon demon)`

Inherited Methods from `IlcConstraint`
`getImpl, getName, getObject, getParentDemon, getSolver, isFalse, isTrue, setName, setObject`

Inherited Methods from `IlcDemon`
`getConstraint, getImpl, getSolver, operator=`

Methods

`public IlcInt` **`getCopiedVarInProcess`**`() const`

This function returns the index of the copied variable which is in process. Indices are described in the class IlcSoftCtHandler.

`public IlcSoftConstraintI *` **`getImpl`**`() const`

This member function returns the implementation object of the invoking object. You can use this member function to check whether a constraint is empty.

`public IlcIntVar` **`getStatusVar`**`()`

This function returns the status variable of the constraint.

`public IlcBool` **`isFailed`**`()`

This function returns `IlcTrue` if the copied variable is violated, otherwise it returns `IlcFalse`.

`public void` **`whenDomainReduction`**`(IlcDemon demon)`

This function is used to link a demon that will be executed each time a copied variable is modified. The current modified copied variable can be accessed using the member function `IlcInt IlcSoftConstraint::getCopiedVarInProcess()const`.

`public void` **`whenFail`**`(IlcDemon demon)`

This function is used to link a demon to the event corresponding to the violation of the copied constraint.

Class IlcSoftCtHandler

Definition file: ilsolver/disjunct.h

The class `IlcSoftCtHandler` is used to manage the definition of a soft constraint. It contains information about the copied constraint, the copied variables and their relation to the original constraint and the original variables.

The parameter `sh` of type `IlcSoftCtHandler` is essential to understand the strength of the mechanism provided by IBM® ILOG® Solver.

The parameter `sh` is filled with some information when a soft constraint is created. This is quite useful to centralize some information when several soft constraints with the same `IlcSoftCtHandler` as argument.

`IlcSoftCtHandler` contains the following information:

- all the original variables
- all the copied variables
- all the original constraints
- all the soft constraints (i.e. all constraints created by the same `IlcSoftHandler` parameter)
- the original variable corresponding to a copied variable
- the original constraint corresponding to a soft constraint
- the variables on which constraints are defined
- the constraints in which a variable is involved

All the functions of this class work with indices. This eases the access to additional data that the user would like to link to each original variable or to each original constraint.

Constructor Summary	
public	`IlcSoftCtHandler()`
public	`IlcSoftCtHandler(IlcSoftCtHandlerI * impl)`
public	`IlcSoftCtHandler(IloSolver solver, IlcInt sizeMax)`

Method Summary	
public IlcConstraint	`getConstraint(const IlcInt ct) const`
public IlcIntVar	`getCopiedVar(const IlcInt cvar) const`
public IlcInt	`getCtOfSoftCt(const IlcInt softCt) const`
public IlcInt	`getFirstCopiedVarOfSoftCt(const IlcInt softCt) const`
public IlcInt	`getFirstCopiedVarOfVar(const IlcInt var) const`
public IlcSoftCtHandlerI *	`getImpl() const`
public IlcInt	`getNextCopiedVarOfSoftCt(const IlcInt softCt, const IlcInt cvar) const`
public IlcInt	`getNextCopiedVarOfVar(const IlcInt var, const IlcInt cvar) const`
public IlcInt	`getNumCt() const`
public IlcInt	`getNumCvars() const`
public IlcInt	`getNumVars() const`
public IlcSoftConstraint	`getSoftConstraint(const IlcInt softCt) const`

public IlcInt	getSoftCtOfCopiedVar(const IlcInt cvar) const
public IlcInt	getSoftCtOfCt(const IlcInt ct) const
public IloSolver	getSolver() const
public IlcIntVar	getVar(const IlcInt var) const
public IlcInt	getVarOfCopiedVar(const IlcInt cvar) const
public void	operator=(const IlcSoftCtHandler & h)

Constructors

public **IlcSoftCtHandler**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IlcSoftCtHandler**(IlcSoftCtHandlerI * impl)

This constructor creates a handle object from a pointer to an implementation object.

public **IlcSoftCtHandler**(IloSolver solver, IlcInt sizeMax)

The argument `sizeMax`specifices the maximum number of copied variables that can be handle by the invoked object.

Methods

public IlcConstraint **getConstraint**(const IlcInt ct) const

This member function returns the constraint of index `ct`.

public IlcIntVar **getCopiedVar**(const IlcInt cvar) const

This member function returns the copied variable of index `cvar`.

public IlcInt **getCtOfSoftCt**(const IlcInt softCt) const

This member function returns the index of the constraint from which the soft constraint `softct` has been copied.

public IlcInt **getFirstCopiedVarOfSoftCt**(const IlcInt softCt) const

This member function returns the index of the first copied variable on which the soft constraint `softct` is defined.

public IlcInt **getFirstCopiedVarOfVar**(const IlcInt var) const

This member function returns the index of the first copied variable of the variable corresponding to index `var`.

`public IlcSoftCtHandlerI * **getImpl**() const`

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

`public IlcInt **getNextCopiedVarOfSoftCt**(const IlcInt softCt, const IlcInt cvar) const`

This member function returns the index of the next copied variable of `cvar` on which the soft constraint is defined.

`public IlcInt **getNextCopiedVarOfVar**(const IlcInt var, const IlcInt cvar) const`

This member function returns the index of the next copied variable of the copied variable `cvar` of the variable corresponding to index `var`.

`public IlcInt **getNumCt**() const`

This member function returns the number of constraints contained in the invoked object.

`public IlcInt **getNumCvars**() const`

This member function returns the number of copied variables contained in the invoked object.

`public IlcInt **getNumVars**() const`

This member function returns the number of variables contained in the invoked object.

`public IlcSoftConstraint **getSoftConstraint**(const IlcInt softCt) const`

This member function returns the copied constraint of index `softct`.

`public IlcInt **getSoftCtOfCopiedVar**(const IlcInt cvar) const`

This member function returns the index of soft constraint `softct` involving the copied `var`.

`public IlcInt **getSoftCtOfCt**(const IlcInt ct) const`

This member function returns the index of the soft constraint of the constraint `ct`.

`public IloSolver **getSolver**() const`

This member function returns an instance of `IloSolver` associated with the invoking object.

`public IlcIntVar **getVar**(const IlcInt var) const`

This member function returns the variable of index `var`.

`public IlcInt **getVarOfCopiedVar**(const IlcInt cvar) const`

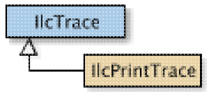
This member function returns the index of the original variable from which the copied variable has been copied.

`public void **operator=**(const IlcSoftCtHandler & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IlcTrace

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>



An instance of this class is part of the Solver trace mechanism. See the virtual member functions of the class `IlcTraceI` to get an idea of what a trace can do in your application.

See Also: IlcPrintTrace, IlcTraceI

Constructor and Destructor Summary	
`public`	`IlcTrace(IlcTraceI * impl)`

Constructors and Destructors

`public` **`IlcTrace`**`(IlcTraceI * impl)`

This constructor creates a handle (an instance of the class `IlcTrace`) from a pointer to an implementation object (an instance of the implementation class `IlcTraceI`).

Class IlcTraceI

Definition file: ilsolver/ilctrace.h
Include file: <ilsolver/ilctrace.h>

An instance of this class is part of the Solver trace mechanism. It enables you to trace such events as failure of a variable, a demon, or a constraint in a solution search. With an instance of this class, you can access the modifications of an active demon or an active constraint at any time by calling the member functions:

- `IlcTraceI::getActiveDemon`
- `IloSolver::getActiveDemon`
- `IloSolver::getActiveGoal`.

Trace Events

Solver uses `#define` to define the events that you can trace with an instance of `IlcTraceI`:

```
#define IlcTraceDemons      ((IlcUInt)  1)
#define IlcTraceConstraint  ((IlcUInt)  2)
#define IlcTraceProcess     ((IlcUInt)  4)
#define IlcTraceVars        ((IlcUInt)  8)
#define IlcTraceFail        ((IlcUInt)  16)
#define IlcTraceNoEvent     ((IlcUInt)  32)
#define IlcTraceAllEvent    ((IlcUInt)  31)
```

When you use its constructor to create an instance of `IlcTraceI`, use one of those values to indicate which event or events to trace. After you have created a trace (an instance of `IlcTraceI`) if you want to alter the events that it traces, use the member functions `setTraceDemon`, `setTraceConstraint`, etc., to alter the trace.

See Also: IlcConstraint, IlcDemon, IlcPrintTrace, IlcTrace

Constructor and Destructor Summary	
public	`IlcTraceI(IloSolver s, IlcUInt flags, const char * name=0)`

Method Summary	
public virtual void	`beginAddRequired(const IlcIntSetVar var, IlcInt val)`
public virtual void	`beginBoolVarProcess(const IlcConstraint constraint)`
public virtual void	`beginConstraintPost(const IlcConstraint constraint)`
public virtual void	`beginConstraintPropagate(const IlcConstraint constraint)`
public virtual void	`beginDemonProcess(const IlcDemon demon)`
public virtual void	`beginFloatVarProcess(const IlcFloatExp exp)`
public virtual void	`beginIntSetVarProcess(const IlcIntSetVar var)`
public virtual void	`beginIntVarProcess(const IlcIntExp exp)`
public virtual void	`beginRemovePossible(const IlcIntSetVar var, IlcInt val)`
public virtual void	`beginRemoveValueFloatVar(const IlcFloatExp, IlcFloat val)`
public virtual void	`beginRemoveValueIntVar(const IlcIntExp exp, IlcInt val)`
public virtual void	`beginSetMaxFloatVar(const IlcFloatExp exp, IlcFloat max)`
public virtual void	`beginSetMaxIntVar(const IlcIntExp exp, IlcInt max)`

public virtual void	beginSetMinFloatVar(const IlcFloatExp exp, IlcFloat min)
public virtual void	beginSetMinIntVar(const IlcIntExp exp, IlcInt min)
public virtual void	beginSetValueFloatVar(const IlcFloatExp, IlcFloat val)
public virtual void	beginSetValueIntVar(const IlcIntExp exp, IlcInt val)
public virtual void	defineIlcOr(IlcInt nbOr)
public virtual void	endAddRequired(const IlcIntSetVar var, IlcInt val)
public virtual void	endBoolVarProcess(const IlcConstraint constraint)
public virtual void	endConstraintPost(const IlcConstraint constraint)
public virtual void	endConstraintPropagate(const IlcConstraint constraint)
public virtual void	endDemonProcess(const IlcDemon demon)
public virtual void	endFloatVarProcess(const IlcFloatExp exp)
public virtual void	endIntSetVarProcess(const IlcIntSetVar var)
public virtual void	endIntVarProcess(const IlcIntExp exp)
public virtual void	endRemovePossible(const IlcIntSetVar var, IlcInt val)
public virtual void	endRemoveValueFloatVar(const IlcFloatExp exp, IlcFloat val)
public virtual void	endRemoveValueIntVar(const IlcIntExp exp, IlcInt val)
public virtual void	endSetMaxFloatVar(const IlcFloatExp exp, IlcFloat max)
public virtual void	endSetMaxIntVar(const IlcIntExp exp, IlcInt max)
public virtual void	endSetMinFloatVar(const IlcFloatExp exp, IlcFloat min)
public virtual void	endSetMinIntVar(const IlcIntExp exp, IlcInt min)
public virtual void	endSetValueFloatVar(const IlcFloatExp exp, IlcFloat val)
public virtual void	endSetValueIntVar(const IlcIntExp exp, IlcInt val)
public virtual void	failDemon(const IlcDemon demon)
public virtual void	failFloatVar(const IlcFloatExp exp)
public virtual void	failIntSetVar(const IlcIntSetVar var)
public virtual void	failIntVar(const IlcIntExp exp)
public virtual void	failManager(IlcInt nbFails)
public IlcDemon	getActiveDemon()
public ostream &	getStream() const
public IlcBool	isAllTraced() const
public IlcBool	isConstraintTraced() const
public IlcBool	isDemonTraced() const
public IlcBool	isFailTraced() const
public IlcBool	isNoneTraced() const
public IlcBool	isProcessTraced() const
public IlcBool	isVarTraced() const
public void	setTraceAllEvents()
public void	setTraceConstraint(IlcBool condition)
public void	setTraceDemons(IlcBool condition)
public void	setTraceFail(IlcBool condition)
public void	setTraceNoEvent()

public void	setTraceProcess(IlcBool condition)
public void	setTraceVars(IlcBool condition)
public void	trace(IlcAnySetVar) const
public void	trace(IlcIntSetVar) const
public void	trace(IlcFloatVar) const
public void	trace(IlcAnyVar) const
public void	trace(IlcIntVar) const

Constructors and Destructors

public **IlcTraceI**(IloSolver s, IlcUInt flags, const char * name=0)

This constructor creates a trace for the solver `s`. The parameter `flags` indicates which events should be traced.

Methods

public virtual void **beginAddRequired**(const IlcIntSetVar var, IlcInt val)

Solver calls this member function when it begins to add the element `val` to the set of required elements of the set variable `var`. (In this context, set means a group or collection of elements.) If the modification does not alter `var`, then Solver does not call this member function.

If you want to use this facility, you must redefine this virtual member function.

public virtual void **beginBoolVarProcess**(const IlcConstraint constraint)

Solver calls this member function when it begins to process a Boolean variable (that is, when the Boolean variable is in process). If the modification does not alter the Boolean variable, then Solver does not call this member function.

If you want to use this facility, you must define this virtual member function.

public virtual void **beginConstraintPost**(const IlcConstraint constraint)

Solver calls this member function when it begins to post `constraint`.

If you want to use this facility, you must define this virtual member function.

public virtual void **beginConstraintPropagate**(const IlcConstraint constraint)

Solver calls this member function when it begins to propagate the constraint `constraint` (that is, immediately before it calls the propagate member function for the constraint).

If you want to use this facility, you must define this virtual member function.

public virtual void **beginDemonProcess**(const IlcDemon demon)

Solver calls this member function when it begins to process `demon` (that is, immediately before it propagates `demon`).

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`beginFloatVarProcess`**`(const IlcFloatExp exp)`

Solver calls this member function when it begins to process a floating-point expression (that is, when the floating-point expression is in process).

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`beginIntSetVarProcess`**`(const IlcIntSetVar var)`

Solver calls this member function when it begins to process an integer set variable, that is, when the integer set variable is in process. In this context, set means a group or collection of elements.

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`beginIntVarProcess`**`(const IlcIntExp exp)`

Solver calls this member function when it begins to process an integer expression (that is, when the integer expression is in process).

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`beginRemovePossible`**`(const IlcIntSetVar var, IlcInt val)`

Solver calls this member function immediately before it removes `val` from the set of possible elements of the integer set variable `var`. (In this context, set means a group or collection of elements.) In other words, `var` has not yet been modified when it is passed to this member function. Normally, if the modification does not alter `var`, then Solver does not call this member function. However, this modification necessarily modifies `var`.

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`beginRemoveValueFloatVar`**`(const IlcFloatExp, IlcFloat val)`

Solver calls this member function immediately before it removes `val` from the domain of `exp`. In other words, `exp` has not yet been modified when it is passed to this member function. Normally, if the modification does not alter `exp`, then Solver does not call this member function. However, this modification necessarily modifies `exp`.

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`beginRemoveValueIntVar`**`(const IlcIntExp exp, IlcInt val)`

Solver calls this member function immediately before it removes `val` from the domain of `exp`. In other words, `exp` has not yet been modified when it is passed to this member function. Normally, if the modification does not alter `exp`, then Solver does not call this member function. However, this modification necessarily modifies `exp`.

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`beginSetMaxFloatVar`**`(const IlcFloatExp exp, IlcFloat max)`

Solver calls this member function immediately before it sets `max` as the maximum of `exp`. (In this context, set means to assign.) In other words, `exp` has not yet been modified when it is passed to this member function. Normally, if the modification does not alter `exp`, then Solver does not call this member function. However, this modification necessarily alters `exp`.

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`beginSetMaxIntVar`**`(const IlcIntExp exp, IlcInt max)`

Solver calls this member function immediately before it sets `max` as the maximum of `exp`. (In this context, set means to assign.) In other words, `exp` has not yet been modified when it is passed to this member function. Normally, if the modification does not alter `exp`, then Solver does not call this member function. However, this modification necessarily alters `exp`.

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`beginSetMinFloatVar`**`(const IlcFloatExp exp, IlcFloat min)`

Solver calls this member function immediately before it sets `min` as the minimum of `exp`. (In this context, set means to assign.) In other words, `exp` has not yet been modified when it is passed to this member function. Normally, if the modification does not alter `exp`, then Solver does not call this member function. However, this modification necessarily alters `exp`.

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`beginSetMinIntVar`**`(const IlcIntExp exp, IlcInt min)`

Solver calls this member function immediately before it sets `min` as the minimum of `exp`. (In this context, set means to assign.) In other words, `exp` has not yet been modified when it is passed to this member function. Normally, if the modification does not alter `exp`, then Solver does not call this member function. However, this modification necessarily alters `exp`.

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`beginSetValueFloatVar`**`(const IlcFloatExp, IlcFloat val)`

Solver calls this member function immediately after it sets `val` as the value of `exp` (that is, after it instantiates `exp`). (In this context, set means to assign.) If the modification does not alter `exp`, then Solver does not call this member function.

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`beginSetValueIntVar`**`(const IlcIntExp exp, IlcInt val)`

Solver calls this member function when it sets `val` as the value of `exp`. (In this context, set means to assign.) That is, Solver calls this member function immediately before it instantiates `exp`. In other words, `exp` has not yet been modified when it is passed to this member function. Normally, if the modification does not alter `exp`, then Solver does not call this member function. However, instantiation necessarily modifies `exp`.

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`defineIlcOr`**`(IlcInt nbOr)`

Solver calls this member function when it sets a choice point.

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`endAddRequired`**`(const IlcIntSetVar var, IlcInt val)`

Solver calls this member function immediately after it adds the element `val` to the set of required elements of the set variable `var`. (In this context, set means a group or collection of elements.) If the modification does not alter `var`, then Solver does not call this member function.

If you want to use this facility, you must redefine this virtual member function.

`public virtual void` **`endBoolVarProcess`**`(const IlcConstraint constraint)`

Solver calls this member function immediately after the Boolean variable indicated by `constraint` is processed.

If you want to use this facility, you must define this virtual member function

`public virtual void` **`endConstraintPost`**`(const IlcConstraint constraint)`

Solver calls this member function immediately after it calls the `post` member function for `constraint` (that is, while `constraint` is in process).

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`endConstraintPropagate`**`(const IlcConstraint constraint)`

Solver calls this member function immediately after it calls the `propagate` member function for the constraint `constraint` (that is, while the constraint is in process).

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`endDemonProcess`**`(const IlcDemon demon)`

Solver calls this member function immediately after it calls the `propagate` member function for `demon` (that is, while `demon` is in process).

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`endFloatVarProcess`**`(const IlcFloatExp exp)`

Solver calls this member function immediately after `exp` is processed.

If you want to use this facility, you must define this virtual member function `public virtual void endIntSetVarProcess(const IlcIntSetVar var)`

Solver calls this member function immediately after the integer set variable `var` is processed. (In this context, set means a group or collection of elements.)

If you want to use this facility, you must define this virtual member function.

`public virtual void endIntVarProcess(const IlcIntExp exp)`

Solver calls this member function immediately after `exp` is processed. If the modification does not alter `exp`, then Solver does not call this member function.

If you want to use this facility, you must define this virtual member function.

`public virtual void endRemovePossible(const IlcIntSetVar var, IlcInt val)`

Solver calls this member function immediately after it removes `val` from the set of possible elements of the integer set variable `var`. (In this context, set means a group or collection of elements.) If the modification does not alter `var`, then Solver does not call this member function.

If you want to use this facility, you must define this virtual member function.

`public virtual void endRemoveValueFloatVar(const IlcFloatExp exp, IlcFloat val)`

Solver calls this member function immediately after it removes `val` from the domain of `exp`. If the modification does not alter `exp`, then Solver does not call this member function.

If you want to use this facility, you must define this virtual member function.

`public virtual void endRemoveValueIntVar(const IlcIntExp exp, IlcInt val)`

Solver calls this member function immediately after it removes `val` from the domain of `exp`. If the modification does not alter `exp`, then Solver does not call this member function.

If you want to use this facility, you must define this virtual member function.

`public virtual void endSetMaxFloatVar(const IlcFloatExp exp, IlcFloat max)`

Solver calls this member function immediately after it sets `max` as the maximum of `exp`. (In this context, set means to assign.) If the modification does not alter `exp`, then Solver does not call this member function.

If you want to use this facility, you must define this virtual member function.

`public virtual void endSetMaxIntVar(const IlcIntExp exp, IlcInt max)`

Solver calls this member function immediately after it sets `max` as the maximum of `exp`. (In this context, set means to assign.) If the modification does not alter `exp`, then Solver does not call this member function.

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`endSetMinFloatVar`**`(const IlcFloatExp exp, IlcFloat min)`

Solver calls this member function immediately after it sets `min` as the minimum of `exp`. (In this context, set means to assign.) If the modification does not alter `exp`, then Solver does not call this member function.

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`endSetMinIntVar`**`(const IlcIntExp exp, IlcInt min)`

Solver calls this member function immediately after it sets `min` as the minimum of `exp`. (In this context, set means to assign.) If the modification does not alter `exp`, then Solver does not call this member function.

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`endSetValueFloatVar`**`(const IlcFloatExp exp, IlcFloat val)`

Solver calls this member function immediately after it sets `val` as the value of `exp` (that is, after it instantiates `exp`). (In this context, set means to assign.) If the modification does not alter `exp`, then Solver does not call this member function.

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`endSetValueIntVar`**`(const IlcIntExp exp, IlcInt val)`

Solver calls this member function immediately after it sets `val` as the value of `exp` (that is, after it instantiates `exp`). (In this context, set means to assign.) If the modification does not alter `exp`, then Solver does not call this member function.

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`failDemon`**`(const IlcDemon demon)`

Solver calls this member function when `demon` triggers a failure.

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`failFloatVar`**`(const IlcFloatExp exp)`

Solver calls this member function when the modification of `exp` triggers a failure.

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`failIntSetVar`**`(const IlcIntSetVar var)`

Solver calls this member function when the modification of the integer set variable `var` triggers a failure. (In this context, set means a group or collection of elements.)

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`failIntVar`**`(const IlcIntExp exp)`

Solver calls this member function when the modification of `exp` triggers a failure. You can access the demon or constraint where `exp` triggered the failure; to do so, use the member function `IlcTraceI::getActiveDemon`.

If you want to use this facility, you must define this virtual member function.

`public virtual void` **`failManager`**`(IlcInt nbFails)`

Solver calls this member function after it calls the member function `IlcGoal::fail` or `IlcConstraint::fail` (that is, when a failure occurs). You can access the current number of failures through the parameter `nbFails`.

`public IlcDemon` **`getActiveDemon`**`()`

This member function returns the demon that is currently active (if there is one). It will return an empty handle if there is no active demon.

Since a constraint is also a demon (that is, `IlcConstraint` derives from `IlcDemon`), you can also use this member function to access the currently active constraint. You may also access any constraint associated with an active demon in these ways:

```
IlcDemon::getConstraint();
```

or

```
getActiveDemon().getConstraint();
```

`public ostream &` **`getStream`**`() const`

This member function indicates which stream will be used for output of the trace.

`public IlcBool` **`isAllTraced`**`() const`

This member function indicates whether all events are being traced.

`public IlcBool` **`isConstraintTraced`**`() const`

This member function indicates whether calls to post and propagate constraints are being traced.

`public IlcBool` **`isDemonTraced`**`() const`

This member function indicates whether calls to propagate demons are being traced.

`public IlcBool` **`isFailTraced`**`() const`

This member function indicates whether failures are being traced.

`public IlcBool` **`isNoneTraced`**`() const`

This member function indicates whether no events are being traced.

`public IlcBool` **`isProcessTraced`**`() const`

This member function indicates whether the variables in process are being traced.

`public IlcBool` **`isVarTraced`**`() const`

This member function indicates whether all variables are being traced.

`public void` **`setTraceAllEvents`**`()`

This member function traces all events.

`public void` **`setTraceConstraint`**`(IlcBool condition)`

If a constraint is pushed onto the propagation queue, then this member function traces it if `condition` is true. In general, this member function traces the posting and propagation of constraints.

`public void` **`setTraceDemons`**`(IlcBool condition)`

This member function traces all calls to all demons, including constraints, if `condition` is true. (A constraint is also a demon.)

`public void` **`setTraceFail`**`(IlcBool condition)`

This member function traces failures if `condition` is true. (For users of previous versions of Solver, this member function replaces `setFailHook`.)

`public void` **`setTraceNoEvent`**`()`

This member function traces no events. That is, it turns off event tracing.

`public void` **`setTraceProcess`**`(IlcBool condition)`

This member function traces all the variables that are in process if `condition` is true.

`public void` **`setTraceVars`**`(IlcBool condition)`

This member function traces the modifications of variables if `condition` is true.

`public void` **`trace`**`(IlcAnySetVar) const`

This member function hooks a variable; that is, it offers a link between a trace and a variable. (For users of previous versions of Solver, this member function resembles the trace hook mechanism.)

`public void` **`trace`**`(IlcIntSetVar) const`

This member function hooks a variable; that is, it offers a link between a trace and a variable. (For users of previous versions of Solver, this member function resembles the trace hook mechanism.)

`public void` **`trace`**`(IlcFloatVar) const`

This member function hooks a variable; that is, it offers a link between a trace and a variable. (For users of previous versions of Solver, this member function resembles the trace hook mechanism.)

`public void` **`trace`**`(IlcAnyVar) const`

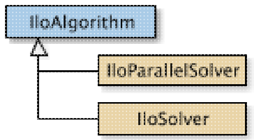
This member function hooks a variable; that is, it offers a link between a trace and a variable. (For users of previous versions of Solver, this member function resembles the trace hook mechanism.)

`public void` **`trace`**`(IlcIntVar) const`

This member function hooks a variable; that is, it offers a link between a trace and a variable. (For users of previous versions of Solver, this member function resembles the trace hook mechanism.)

Class IloAlgorithm

Definition file: ilconcert/iloalg.h



The base class of algorithms in Concert Technology.
An instance of `IloAlgorithm` represents an algorithm in Concert Technology.

In general terms, you define a model, and Concert Technology extracts objects from it for your target algorithm and then solves for solutions.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Status

The member function `getStatus` returns a status showing information about the currently extracted model and the solution (if there is one). For explanations of the status, see the nested enumeration `IloAlgorithm::Status`.

Exceptions

The class `IloAlgorithm::Exception`, derived from the class `IloException`, is the base class of exceptions thrown by classes derived from `IloAlgorithm`. For an explanation of exceptions thrown by instances of `IloAlgorithm`, see `IloAlgorithm::Exception`.

Streams and Output

The class `IloAlgorithm` supports these communication streams:

- `ostream& IloAlgorithm::error() const;` for error messages.
- `ostream& IloAlgorithm::out() const;` for general output.
- `ostream& IloAlgorithm::warning() const;` for warning messages about nonfatal conditions.

Child classes:

- the class `IloCplex` in the *IBM ILOG CPLEX Reference Manual*
- the class `IloCP` in the *IBM ILOG CP Optimizer Reference Manual.*
- the class `IloSolver` in the *IBM ILOG Solver Reference Manual* .

See Also: IloEnv, IloModel, IloAlgorithm::Status, IloAlgorithm::Exception

Constructor Summary	
`public`	`IloAlgorithm(IloAlgorithmI * impl=0)`

Method Summary	
`public void`	`clear() const`
`public void`	`end()`
`public ostream &`	`error() const`
`public void`	`extract(const IloModel) const`
`public IloEnv`	`getEnv() const`

public IloInt	getIntValue(const IloIntVar) const
public void	getIntValues(const IloIntVarArray, IloIntArray) const
public IloModel	getModel() const
public IloNum	getObjValue() const
public IloAlgorithm::Status	getStatus() const
public IloNum	getTime() const
public IloNum	getValue(const IloNumExprArg) const
public IloNum	getValue(const IloObjective) const
public IloNum	getValue(const IloIntVar) const
public IloNum	getValue(const IloNumVar) const
public void	getValues(const IloIntVarArray, IloNumArray) const
public void	getValues(const IloNumVarArray, IloNumArray) const
public IloBool	isExtracted(const IloExtractable) const
public ostream &	out() const
public void	printTime() const
public void	resetTime() const
public void	setError(ostream &)
public void	setOut(ostream &)
public void	setWarning(ostream &)
public IloBool	solve() const
public ostream &	warning() const

Inner Enumeration	
IloAlgorithm::Status	An enumeration for the class `IloAlgorithm`.

Inner Class	
IloAlgorithm::CannotExtractException	The class of exceptions thrown if an object cannot be extracted from a model.
IloAlgorithm::CannotRemoveException	The class of exceptions thrown if an object cannot be removed from a model.
IloAlgorithm::Exception	The base class of exceptions thrown by classes derived from IloAlgorithm.
IloAlgorithm::NotExtractedException	The class of exceptions thrown if an extractable object has no value in the current solution of an algorithm.

Constructors

public **IloAlgorithm**(IloAlgorithmI * impl=0)

This constructor creates an algorithm in Concert Technology from its implementation object. This is the default constructor.

Methods

`public void **clear**() const`

This member function clears the current model from the algorithm.

`public void **end**()`

This member function deletes the invoking algorithm. That is, it frees memory associated with the invoking algorithm.

`public ostream & **error**() const`

This member function returns a reference to the stream currently used for error messages from the invoking algorithm. `IloAlgorithm::error` is initialized with the value of `IloEnv::error`.

`public void **extract**(const IloModel) const`

This member function extracts the extractable objects from a model into the invoking algorithm if a member function exists to extract the objects from the model for the invoking algorithm. Not all extractable objects can be extracted by all algorithms; see the documentation of the algorithm class you are using for a list of extractable classes it supports.

When you use this member function to extract extractable objects from a model, it extracts all the elements of that model for which Concert Technology creates the representation of the extractable object suitable for the invoking algorithm.

The attempt to extract may fail. In case such a failure occurs, Concert Technology throws the exception `CannotExtractException` on platforms that support C++ exceptions when exceptions are enabled.

For example, a failure will occur if you attempt to extract more than one objective for an invoking algorithm that accepts only one objective, and Concert Technology will throw the exception `MultipleObjException`.

`public IloEnv **getEnv**() const`

This member function returns the environment of the invoking algorithm.

`public IloInt **getIntValue**(const IloIntVar) const`

This member function returns the integer value of an integer variable in the current solution of the invoking algorithm. For example, to access the variable, use the member function `getIntValue(var)` where `var` is an instance of the class `IloIntVar`.

If there is no value to return for `var`, this member function raises an error. This member function throws the exception `NotExtractedException` if there is no value to return (for example, if `var` was not extracted by the invoking algorithm).

`public void **getIntValues**(const IloIntVarArray, IloIntArray) const`

This member function accepts an array of variables `vars` and puts the corresponding values into the array `vals`; the corresponding values come from the current solution of the invoking algorithm. The array `vals` must be a clean, empty array when you pass it to this member function.

If there are no values to return for `vars`, this member function raises an error. On platforms that support C++ exceptions, when exceptions are enabled, this member function throws the exception `NotExtractedException` in such a case.

`public IloModel` **`getModel`**`() const`

This member function returns the model of the invoking algorithm.

`public IloNum` **`getObjValue`**`() const`

This member function returns the numeric value of the objective function associated with the invoking algorithm.

`public IloAlgorithm::Status` **`getStatus`**`() const`

This member function returns a status showing information about the current model and the solution. For explanations of the status, see the nested enumeration `IloAlgorithm::Status`.

`public IloNum` **`getTime`**`() const`

This member function returns the amount of time elapsed in seconds since the most recent reset of the invoking algorithm. (The member function `IloAlgorithm::printTime` directs the output of `getTime` to the output channel of the invoking algorithm.)

See Also: IloTimer

`public IloNum` **`getValue`**`(const IloNumExprArg) const`

This member function returns the value of an expression in the current solution of the invoking algorithm. For example, to access the expression, use the member function `getValue(expr)` where `expr` is an instance of the class `IloNumExprArg`.

If there is no value to return for `expr`, this member function raises an error. This member function throws the exception `NotExtractedException` if there is no value to return (for example, if `expr` was not extracted by the invoking algorithm).

`public IloNum` **`getValue`**`(const IloObjective) const`

This member function returns the value of an objective in the current solution of the invoking algorithm. For example, to access the objective, use the member function `getValue(obj)` where `obj` is an instance of the class `IloObjective`.

If there is no value to return for `obj`, this member function raises an error. This member function throws the exception `NotExtractedException` if there is no value to return (for example, if `obj` was not extracted by the invoking algorithm).

`public IloNum **getValue**(const IloIntVar) const`

This member function returns the numeric value of an integer variable in the current solution of the invoking algorithm. For example, to access the variable, use the member function `getValue(var)` where `var` is an instance of the class `IloIntVar`.

If there is no value to return for `var`, this member function raises an error. This member function throws the exception `NotExtractedException` if there is no value to return (for example, if `var` was not extracted by the invoking algorithm).

`public IloNum **getValue**(const IloNumVar) const`

This member function returns the numeric value of a numeric variable in the current solution of the invoking algorithm. For example, to access the value of the variable, use the member function `getValue(var)` where `var` is an instance of the class `IloNumVar`.

If there is no value to return for `var`, this member function raises an error. This member function throws the exception `NotExtractedException` if there is no value to return (for example, if `var` was not extracted by the invoking algorithm).

`public void **getValues**(const IloIntVarArray, IloNumArray) const`

This member function accepts an array of variables `vars` and puts the corresponding values into the array `vals`; the corresponding values come from the current solution of the invoking algorithm. The array `vals` must be a clean, empty array when you pass it to this member function.

If there are no values to return for `vars`, this member function raises an error. On platforms that support C++ exceptions, when exceptions are enabled, this member function throws the exception `NotExtractedException` in such a case.

`public void **getValues**(const IloNumVarArray, IloNumArray) const`

This member function accepts an array of variables `vars` and puts the corresponding values into the array `vals`; the corresponding values come from the current solution of the invoking algorithm. The array `vals` must be a clean, empty array when you pass it to this member function.

If there are no values to return for `vars`, this member function raises an error. On platforms that support C++ exceptions, when exceptions are enabled, this member function throws the exception `NotExtractedException` in such a case.

`public IloBool **isExtracted**(const IloExtractable) const`

This member function returns `IloTrue` if `extr` has been extracted for the invoking algorithm; otherwise, it returns `IloFalse`.

`public ostream & **out**() const`

This member function returns a reference to the stream currently used for logging. General output from the invoking algorithm is accessible through this member function. `IloAlgorithm::out` is initialized with the value of `IloEnv::out`.

`public void printTime() const`

This member function directs the output of the member function `IloAlgorithm::getTime` to an output channel of the invoking algorithm. (The member function `IloAlgorithm::getTime` accesses the elapsed time in seconds since the most recent reset of the invoking algorithm.)

`public void resetTime() const`

This member function resets the timer on the invoking algorithm. The type of timer is platform dependent. On Windows systems, the time is elapsed wall clock time. On UNIX systems, the time is CPU time.

`public void setError(ostream &)`

This member function sets the stream for errors generated by the invoking algorithm. By default, the stream is defined by an instance of `IloEnv` as `cerr`.

`public void setOut(ostream &)`

This member function redirects the `out()` stream with the stream given as an argument.

This member function can be used with `IloEnv::getNullStream` to suppress screen output by redirecting it to the null stream.

`public void setWarning(ostream &)`

This member function sets the stream for warnings from the invoking algorithm. By default, the stream is defined by an instance of `IloEnv` as `cout`.

`public IloBool solve() const`

This member function solves the current model in the invoking algorithm. In other words, `solve` works with all extractable objects extracted from the model for the algorithm. The member function returns `IloTrue` if it finds a solution (not necessarily an optimal one). Here is an example of its use:

```
if (algo.solve()) {
  algo.out() << "Status is " << algo.getStatus() << endl;
};
```

If an objective of the model has been extracted into the invoking algorithm, this member function solves the model to optimality. If there is currently no objective, this member function searches for the first feasible solution. A feasible solution is not necessarily optimal, though it satisfies all constraints.

`public ostream & warning() const`

This member function returns a reference to the stream currently used for warnings from the invoking algorithm. `IloAlgorithm::warning` is initialized with the value of `IloEnv::warning`.

Inner Enumerations

Enumeration Status

Definition file: ilconcert/iloalg.h

An enumeration for the class `IloAlgorithm`.
`IloAlgorithm` is the base class of algorithms in Concert Technology, and `IloAlgorithm::Status` is an enumeration limited in scope to the class `IloAlgorithm`. The member function `IloAlgorithm::getStatus` returns a status showing information about the current model and the solution.

`Unknown` specifies that the algorithm has no information about the solution of the model.

`Feasible` specifies that the algorithm found a feasible solution (that is, an assignment of values to variables that satisfies the constraints of the model, though it may not necessarily be optimal). The member functions `IloAlgorithm::getValue` access this feasible solution.

`Optimal` specifies that the algorithm found an optimal solution (that is, an assignment of values to variables that satisfies all the constraints of the model and that is proved optimal with respect to the objective of the model). The member functions `IloAlgorithm::getValue` access this optimal solution.

`Infeasible` specifies that the algorithm proved the model infeasible; that is, it is not possible to find an assignment of values to variables satisfying all the constraints in the model.

`Unbounded` specifies that the algorithm proved the model unbounded.

`InfeasibleOrUnbounded` specifies that the model is infeasible or unbounded.

`Error` specifies that an error occurred and, on platforms that support exceptions, that an exception has been thrown.

See Also: IloAlgorithm, operator<<

Fields:

`Unknown`

`Feasible`

`Optimal`

`Infeasible`

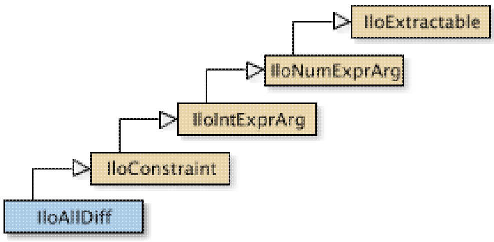
`Unbounded`

`InfeasibleOrUnbounded`

`Error`

Class IloAllDiff

Definition file: ilconcert/ilomodel.h



For constraint programming: constrains integer variables to assume different values in a model.
An instance of this class is a constraint that forces constrained *integer* variables to assume different values from one another in a model. In other words, no two of those integer variables will have the same integer value when this constraint is satisfied.

In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

What Is Extracted

All the variables (that is, instances of IloIntVar or one of its subclasses) that have appeared as an argument of a constructor of `IloAllDiff` and all variables that have been explicitly added to the instance of `IloAllDiff` will be extracted by an algorithm (such as an instance of `IloCP` or `IloSolver`, that extracts that constraint.

`IloCplex` does **not** extract instances of `IloAllDiff`.

See Also: IloAdd, IloConstraint, IloDiff

Constructor Summary	
`public`	`IloAllDiff()`
`public`	`IloAllDiff(IloAllDiffI * impl)`
`public`	`IloAllDiff(const IloEnv env, const IloIntVarArray vars, const char * name=0)`

Method Summary	
`public IloAllDiffI *`	`getImpl() const`

Inherited Methods from `IloConstraint`
`getImpl`

Inherited Methods from `IloIntExprArg`
`getImpl`

Inherited Methods from `IloNumExprArg`
`getImpl`

Inherited Methods from `IloExtractable`
`asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject`

Constructors

`public` **`IloAllDiff`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IloAllDiff`**`(IloAllDiffI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public` **`IloAllDiff`**`(const IloEnv env, const IloIntVarArray vars, const char * name=0)`

This constructor creates a constraint that forces all the integer variables in `vars` to assume different values from each other. If `vars` is empty, this constructor creates an empty instance of `IloAllDiff`, and then you must fill the constraint; that is, you must put variables into the array. You must add this constraint to a model and extract the model for an algorithm in order for it to be taken into account.

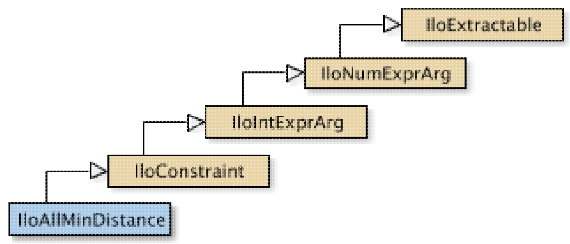
Methods

`public IloAllDiffI *` **`getImpl`**`() const`

This member function returns a pointer to the implementation object of the invoking handle.

Class IloAllMinDistance

Definition file: ilconcert/ilomodel.h



For constraint programming: constraint on the minimum absolute distance between a pair of variables in an array.
An instance of the class `IloAllMinDistance` is a constraint that makes sure that the absolute distance between any pair of variables in an array of constrained integer variables will be greater than or equal to a given integer.

What Is Extracted

All the variables that have been added to the model and that have not been removed from it will be extracted when the algorithm `IloCP` or `IloSolver` extracts the constraint.

`IloCplex` does **not** extract this constraint.

See Also: IloAllDiff

Constructor Summary	
public	IloAllMinDistance()
public	IloAllMinDistance(IloAllMinDistanceI * impl)
public	IloAllMinDistance(const IloEnv env, const IloIntVarArray vars, IloInt k, const char * name=0)

Method Summary	
public IloAllMinDistanceI *	getImpl() const

Inherited Methods from `IloConstraint`
getImpl

Inherited Methods from `IloIntExprArg`
getImpl

Inherited Methods from `IloNumExprArg`
getImpl

Inherited Methods from `IloExtractable`
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Constructors

`public IloAllMinDistance()`

This constructor creates an empty handle. You must initialize it before you use it.

`public IloAllMinDistance(IloAllMinDistanceI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public IloAllMinDistance(const IloEnv env, const IloIntVarArray vars, IloInt k, const char * name=0)`

This constructor returns a constraint that insures that the absolute distance between any pair of variables in the array `vars` will be greater than or equal to `k`. You must add this constraint to a model and extract the model for an algorithm in order for it to be taken into account.

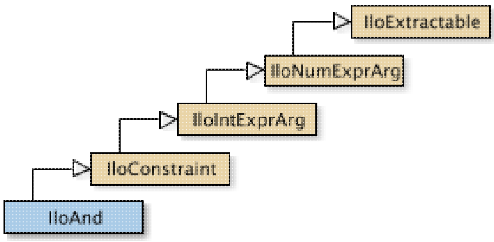
Methods

`public IloAllMinDistanceI * getImpl() const`

This member function returns a pointer to the implementation object of the invoking handle.

Class IloAnd

Definition file: ilconcert/ilomodel.h



Defines a logical conjunctive-AND among other constraints.
An instance of `IloAnd` represents a conjunctive constraint. In other words, it defines a logical conjunctive-AND among any number of constraints. It lets you represent a constraint on constraints in your model. Since an instance of `IloAnd` is a constraint itself, you can build up extensive conjunctions by adding constraints to an instance of `IloAnd` by means of the member function `IloAnd::add`. You can also remove constraints from an instance of `IloAnd` by means of the member function `IloAnd::remove`.

The elements of a conjunctive constraint must be in the same environment.

In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Conjunction of Goals

If you want to represent the conjunction of goals (rather than constraints) in your model, then you should consider the function `IloAndGoal` (documented in the IBM ILOG Solver Reference Manual).

What Is Extracted

All the constraints (that is, instances of `IloConstraint` or one of its subclasses) that have been added to a conjunctive constraint (an instance of `IloAnd`) and that have not been removed from it will be extracted when an algorithm such as `IloCplex`, `IloCP`, or `IloSolver` extracts the constraint.

Example

For example, you may write:

```
IloAnd and(env);
and.add(constraint1);
and.add(constraint2);
and.add(constraint3);
```

Those lines are equivalent to :

```
IloAnd and = constraint1 && constraint2 && constraint3;
```

See Also: IloConstraint, IloOr, operator&&

Constructor Summary	
public	IloAnd()

public	IloAnd(IloAndI * impl)
public	IloAnd(const IloEnv env, const char * name=0)

Method Summary	
public void	add(const IloConstraintArray array) const
public void	add(const IloConstraint constraint) const
public IloAndI *	getImpl() const
public void	remove(const IloConstraintArray array) const
public void	remove(const IloConstraint constraint) const

Inherited Methods from `IloConstraint`
getImpl

Inherited Methods from `IloIntExprArg`
getImpl

Inherited Methods from `IloNumExprArg`
getImpl

Inherited Methods from `IloExtractable`
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Constructors

public **IloAnd**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IloAnd**(IloAndI * impl)

This constructor creates a handle object from a pointer to an implementation object.

public **IloAnd**(const IloEnv env, const char * name=0)

This constructor creates a conjunctive constraint for use in the environment `env`. In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

The optional argument `name` is set to `0` by default.

Methods

public void **add**(const IloConstraintArray array) const

This member function makes all the elements in `array` elements of the invoking conjunctive constraint. In other words, it applies the invoking conjunctive constraint to all the elements of `array`.

Note

The member function `add` notifies Concert Technology algorithms about this change to the invoking object.

`public void **add**(const IloConstraint constraint) const`

This member function makes `constraint` one of the elements of the invoking conjunctive constraint. In other words, it applies the invoking conjunctive constraint to `constraint`.

Note

The member function `add` notifies Concert Technology algorithms about this change to the invoking object.

`public IloAndI * **getImpl**() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public void **remove**(const IloConstraintArray array) const`

This member function removes all the elements of `array` from the invoking conjunctive constraint so that the invoking conjunctive constraint no longer applies to any of those elements.

Note

The member function `remove` notifies Concert Technology algorithms about this change to the invoking object.

`public void **remove**(const IloConstraint constraint) const`

This member function removes `constraint` from the invoking conjunctive constraint so that the invoking conjunctive constraint no longer applies to `constraint`.

Note

The member function `remove` notifies Concert Technology algorithms about this change to the invoking object.

Class IloAnyArray

Definition file: ilconcert/iloany.h

For IBM® ILOG® Solver: array class of the enumerated type definition `IloAny`.
For each basic type, Concert Technology defines a corresponding array class. `IloAnyArray` is the array class of the basic enumerated type definition (`IloAny`) for a model.

Instances of `IloAnyArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

If you would like to represent a set of enumerated values (that is, no repeated elements, no order among elements), consider an instance of `IloAnySet`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloAny, IloAnySet, operator>>, operator<<

Constructor Summary	
`public`	`IloAnyArray(IloDefaultArrayI * i=0)`
`public`	`IloAnyArray(const IloAnyArray & copy)`
`public`	`IloAnyArray(const IloEnv env, IloInt n=0)`
`public`	`IloAnyArray(const IloEnv env, IloInt n, const IloAny p0, const IloAny p1, ...)`

Method Summary	
`public void`	`add(const IloAny p)`
`public IloBool`	`contains(IloAny e) const`

Constructors

public **IloAnyArray**(IloDefaultArrayI * i=0)

This constructor creates an empty array of elements. You cannot create instances of the undocumented class `IloDefaultArrayI`. As an argument in this default constructor, it allows you to pass 0 (zero) as a value to an optional argument in functions and member functions that accept an array as an argument.

public **IloAnyArray**(const IloAnyArray & copy)

This copy constructor creates a handle to the array of pointers specified by `copy`.

public **IloAnyArray**(const IloEnv env, IloInt n=0)

This constructor creates an array of `n` elements, all of which are empty handles.

```
public IloAnyArray(const IloEnv env, IloInt n, const IloAny p0, const IloAny p1,
...)
```

This constructor creates an array of `n` elements for use in a model.

Methods

```
public void add(const IloAny p)
```

This member function appends `p` to the invoking array.

```
public IloBool contains(IloAny e) const
```

This member function checks whether the value is contained or not.

Class IloAnyBinaryPredicate

Definition file: ilconcert/ilotupleset.h

For IBM ILOG Solver: defines binary predicates on objects in a model.
This class makes it possible for you to define binary predicates operating on arbitrary objects in a model. A predicate is an object with a member function (such as `IloAnyBinaryPredicate::isTrue`) that checks whether or not a property is satisfied by an ordered set of (pointers to) objects.

Defining a New Class of Predicates

Predicates, like other Concert Technology objects, depend on two classes: a handle class, `IloAnyBinaryPredicate`, and an implementation class, such as `IloAnyBinaryPredicateI`, where an object of the handle class contains a data member (the handle pointer) that points to an object (its implementation object) of an instance of `IloAnyBinaryPredicateI` allocated in a Concert Technology environment. As a Concert Technology user, you will be working primarily with handles.

If you define a new class of predicates yourself, you must define its implementation class together with the corresponding virtual member function `isTrue`, as well as a member function that returns an instance of the handle class `IloAnyBinaryPredicate`.

Arity

As a developer, you can use predicates in Concert Technology applications to define your own constraints that have not already been predefined in Concert Technology. In that case, the *arity* of the predicate (that is, the number of constrained variables involved in the predicate, and thus the size of the array that the member function `IloAnyBinaryPredicate::isTrue` must check) must be two.

See Also: IloTableConstraint

Constructor and Destructor Summary	
`public`	`IloAnyBinaryPredicate()`
`public`	`IloAnyBinaryPredicate(IloAnyBinaryPredicateI * impl)`

Method Summary	
`public IloAnyBinaryPredicateI *`	`getImpl() const`
`public IloBool`	`isTrue(const IloAny val1, const IloAny val2)`
`public void`	`operator=(const IloAnyBinaryPredicate & h)`

Constructors and Destructors

`public` **`IloAnyBinaryPredicate`**`()`

This constructor creates an empty binary predicate. In other words, the predicate is an empty handle with a null handle pointer. You must assign the elements of the predicate before you attempt to access it, just as you would any other pointer. Any attempt to access it before this assignment will throw an exception (an instance of `IloSolver::SolverErrorException`).

`public` **`IloAnyBinaryPredicate`**`(IloAnyBinaryPredicateI * impl)`

This constructor creates a handle object (an instance of the class `IloAnyBinaryPredicate`) from a pointer to an implementation object (an instance of the implementation class `IlcAnyPredicateI`, documented in the *IBM ILOG Solver Reference Manual*).

Methods

`public IloAnyBinaryPredicateI * **getImpl**() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public IloBool **isTrue**(const IloAny val1, const IloAny val2)`

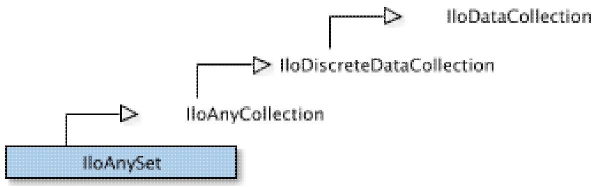
This member function returns `IloTrue` if the values `val1` and `val2` make the invoking binary predicate valid. It returns `IloFalse` otherwise.

`public void **operator=**(const IloAnyBinaryPredicate & h)`

This assignment operator copies `h` into the invoking predicate by assigning an address to the handle pointer of the invoking object. That address is the location of the implementation object of the argument `h`. After execution of this operator, both the invoking predicate and `h` point to the same implementation object.

Class IloAnySet

Definition file: ilconcert/iloanyset.h



A class to represent a set of enumeration values.
An instance of this class represents a set of enumerated values. The same enumerated value will not appear more than once in a set. The elements of a set are not ordered. The class `IloAnySet::Iterator` offers you a way to traverse the elements of such a set.

If you are considering modeling issues where you want to represent repeated elements or where you want to exploit an indexed order among the elements, then you might want to look at the class `IloAnyArray` instead of this class for sets.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloAny, IloAnyArray, IloAnySet::Iterator

Constructor Summary	
`public`	`IloAnySet(const IloEnv env, const IloAnyArray array, IloBool withIndex=IloFalse)`
`public`	`IloAnySet(const IloEnv env, IloBool withIndex=IloFalse)`

Method Summary	
`public void`	`add(IloAnySet set)`
`public void`	`add(IloAny elt)`
`public IloBool`	`contains(IloAnySet set) const`
`public IloBool`	`contains(IloAny elt) const`
`public IloAnySet`	`copy() const`
`public void`	`empty()`
`public IloAny`	`getNext(IloAny value, IloInt n=1) const`
`public IloAny`	`getNextC(IloAny value, IloInt n=1) const`
`public IloAny`	`getPrevious(IloAny value, IloInt n=1) const`
`public IloAny`	`getPreviousC(IloAny value, IloInt n=1) const`
`public IloInt`	`getSize() const`
`public IloBool`	`intersects(IloAnySet set) const`
`public void`	`remove(IloAnySet set)`
`public void`	`remove(IloAny elt)`
`public void`	`setIntersection(IloAnySet set)`
`public void`	`setIntersection(IloAny elt)`

Inner Class

IloAnySet::Iterator	For IBM® ILOG® Solver: an iterator to traverse the elements of `IloAnySet`.

Constructors

`public` **`IloAnySet`**`(const IloEnv env, const IloAnyArray array, IloBool withIndex=IloFalse)`

This constructor creates a set of enumerated values for `env` from the elements in `array`. The optional flag `withIndex` corresponds to the activation or not of internal Hash Tables to improve speed of `add`/`getIndex` methods.

`public` **`IloAnySet`**`(const IloEnv env, IloBool withIndex=IloFalse)`

This constructor creates an empty set (no elements) for `env`. You must use the member function `IloAnySet::add` to fill this set with elements. The optional flag `withIndex` corresponds to the activation or not of internal Hash Tables to improve speed of `add`/`getIndex` methods.

Methods

`public void` **`add`**`(IloAnySet set)`

This member function adds `set` to the invoking set. By adds, we mean that the invoking set becomes the union of its former elements and the elements of `set`.

`public void` **`add`**`(IloAny elt)`

This member function adds `elt` to the invoking set. By adds, we mean that the invoking set becomes the union of its former elements and the new `elt`.

`public IloBool` **`contains`**`(IloAnySet set) const`

This member function returns a Boolean value (zero or one) that specifies whether `set` intersects the invoking set. The value one specifies that the invoking set contains all the elements of `set`, and that the intersection of the invoking set with `set` is precisely set. The value zero specifies that the intersection of the invoking set `set` is not precisely `set`.

`public IloBool` **`contains`**`(IloAny elt) const`

This member function returns a Boolean value (zero or one) that specifies whether `elt` is an element of the invoking set. The value one specifies that the invoking set contains `elt`; the value zero specifies that the invoking set does not contain `elt`.

`public IloAnySet` **`copy`**`() const`

This member functions creates a clone of the array.

`public void` **`empty`**`()`

This member function removes the elements from the invoking set. In other words, the invoking set becomes the empty set.

`public IloAny getNext(IloAny value, IloInt n=1) const`

This method return the value next to the given argument in the set.

If the given value does not exist, it throws an exception

If no value follows, i.e. you are at the end of the set, it throws an exception

See also getNextC, getPreviousC for circular search.

`public IloAny getNextC(IloAny value, IloInt n=1) const`

This method return the value next to the given argument in the set.

If the given value does not exist, it throws an exception

If no value follows, i.e. you are at the end of the set, it will give you the first value (circular search)

See also getNext, getPrevious.

`public IloAny getPrevious(IloAny value, IloInt n=1) const`

This method return the value previous to the given argument in the set.

If the given value does not exist, it throws an exception

If no value is previous, i.e. you are at the beginning of the set, it throws an exception

See also getNextC, getPreviousC for circular search.

`public IloAny getPreviousC(IloAny value, IloInt n=1) const`

This method return the value previous to the given argument in the set.

If the given value does not exist, it throws an exception

If no value is prvious, i.e. you are at the beginning of the set, it will give you the last value (circular search)

See also getNext, getPrevious.

`public IloInt getSize() const`

This member function returns an integer specifying the size of the invoking set (that is, how many elements it contains).

`public IloBool intersects(IloAnySet set) const`

This member function returns a Boolean value (zero or one) that specifies whether `set` intersects the invoking set. The value one specifies that the intersection of `set` and the invoking set is not empty (at least one element in common); the value zero specifies that the intersection of `set` and the invoking set is empty (no elements in common).

`public void` **`remove`**`(IloAnySet set)`

This member function removes all the elements of `set` from the invoking set.

`public void` **`remove`**`(IloAny elt)`

This member function removes `elt` from the invoking set.

`public void` **`setIntersection`**`(IloAnySet set)`

This member function changes the invoking set so that it includes only the elements of `set`. In other words, the invoking set becomes the intersection of its former elements with the elements of `set`.

`public void` **`setIntersection`**`(IloAny elt)`

This member function changes the invoking set so that it includes only the element specified by `elt`. In other words, the invoking set becomes the intersection of its former elements with `elt`.

Class IloAnySetValueSelector

Definition file: ilsolver/ilosolverset.h
Include file: <ilsolver/ilosolver.h>

Solver lets you create value selectors to control the order in which the values in the domain of a set of constrained enumerated variables are tried during the search for a solution.

The class `IloAnySetValueSelector` represents value selectors in an IBM® ILOG® Concert Technology *model*. The class `IlcAnySetSelect` represents value selectors internally in a Solver search.

This class is the handle class of the modeling object that wraps the search object. When search starts, `IloAnySetValueSelectorI` is extracted into an instance of `IlcAnySetSelectI`.

See Also: IlcAnySetSelect, IloAnySetValueSelectorI

Constructor Summary	
public	IloAnySetValueSelector()
public	IloAnySetValueSelector(IloAnySetValueSelectorI * impl)

Method Summary	
public IloAnySetValueSelectorI *	getImpl() const
public void	operator=(const IloAnySetValueSelector & h)

Constructors

`public` **`IloAnySetValueSelector`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IloAnySetValueSelector`**`(IloAnySetValueSelectorI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

Methods

`public IloAnySetValueSelectorI *` **`getImpl`**`() const`

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

`public void` **`operator=`**`(const IloAnySetValueSelector & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IloAnySetValueSelectorI

Definition file: ilsolver/ilosolverset.h
Include file: <ilsolver/ilosolver.h>

This class is the implementation class for `IloAnySetValueSelector`.

The class `IloAnySetValueSelectorI` is the implementation class for value selectors in an IBM® ILOG® Concert Technology *model*. The class `IlcAnySetSelectI` is the implementation class for value selectors internally in a Solver search.

This class is the modeling object that wraps the search object. When search starts, `IloAnySetValueSelectorI` is extracted into an instance of `IlcAnySetSelectI`.

To define new selection criteria, you define both a subclass of `IloAnySetValueSelectorI` and a subclass of `IlcAnySetSelectI`.

See Also: IlcAnySetSelect, IloAnySetValueSelector

Constructor and Destructor Summary	
public	IloAnySetValueSelectorI(IloEnvI *)
public	~IloAnySetValueSelectorI()

Method Summary	
public virtual void	display(ostream &) const
public virtual IlcAnySetSelect	extract(const IloSolver solver) const
public IloEnvI *	getEnv() const
public virtual IloAnySetValueSelectorI *	makeClone(IloEnvI * env) const

Constructors and Destructors

public **IloAnySetValueSelectorI**(IloEnvI *)

This constructor creates an instance of the class `IloAnySetValueSelectorI`. This constructor should not be called directly as this class is an abstract class. This constructor is called automatically in the constructor of its subclasses.

public **~IloAnySetValueSelectorI**()

This destructor is called automatically by the destructor of its subclasses. It frees memory used by the invoking object.

Methods

public virtual void **display**(ostream &) const

This member function prints the invoking value selector on an output stream.

`public virtual IlcAnySetSelect` **`extract`**`(const IloSolver solver) const`

In general terms, in Concert Technology, the objects of a model must be extracted for an algorithm (an instance of one of the subclasses of `IloAlgorithm`, such as `IloSolver`). This member function returns the internal value selector extracted for `solver` from the invoking value selector of a model.

`public IloEnvI *` **`getEnv`**`() const`

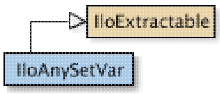
This member function returns the environment to which the invoking value selector belongs. A value selector belongs to exactly one environment; different environments cannot share the same value selector.

`public virtual IloAnySetValueSelectorI *` **`makeClone`**`(IloEnvI * env) const`

This member function is called internally to duplicate the current value selector.

Class IloAnySetVar

Definition file: ilconcert/iloanyset.h



For IBM® ILOG® Solver: a class to represent a set of enumerated values as a constrained variable.
An instance of this class offers a convenient way to represent a set of enumerated values as a constrained variable in Concert Technology.

A constrained variable representing a set of enumerated values (that is, an instance of `IloAnySetVar`) is defined in terms of two other sets: its required elements and its possible elements. Its required elements are those that must be in the set. Its possible elements are those that may be in the set. This class offers member functions for accessing the required and possible elements of a set of enumerated values.

The function `IloCard` offers you a way to constrain the number of elements in a set variable. That is, `IloCard` constrains the cardinality of a set variable.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloAnySet, IloAnySetVarArray, IloCard, IloEqIntersection, IloEqUnion, IloExtractable, IloMember, IloModel, IloNotMember, IloNullIntersect, IloSubset, IloSubsetEq

Constructor Summary	
`public`	`IloAnySetVar()`
`public`	`IloAnySetVar(IloIntSetVarI * impl)`
`public`	`IloAnySetVar(const IloEnv env, const IloAnyArray possible, const char * name=0)`
`public`	`IloAnySetVar(const IloEnv env, const IloAnyArray possible, const IloAnyArray required, const char * name=0)`
`public`	`IloAnySetVar(const IloAnyCollection possible, const char * name=0)`
`public`	`IloAnySetVar(const IloAnyCollection possible, const IloAnyCollection required, const char * name=0)`

Method Summary	
`public void`	`addPossible(IloAny elt) const`
`public void`	`addRequired(IloAny elt) const`
`public IloIntSetVarI *`	`getImpl() const`
`public void`	`getPossibleSet(IloAnySet set) const`
`public IloAnySet`	`getPossibleSet() const`
`public IloAnySet::Iterator`	`getPossibleSetIterator() const`
`public void`	`getRequiredSet(IloAnySet set) const`
`public IloAnySet`	`getRequiredSet() const`
`public IloAnySet::Iterator`	`getRequiredSetIterator() const`
`public void`	`removePossible(IloAny elt) const`
`public void`	`removeRequired(IloAny elt) const`

Inherited Methods from `IloExtractable`
`asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject`

Constructors

`public **IloAnySetVar**()`

This constructor creates an empty handle. You must initialize it before you use it.

`public **IloAnySetVar**(IloIntSetVarI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public **IloAnySetVar**(const IloEnv env, const IloAnyArray possible, const char * name=0)`

This constructor creates a constrained set variable from the values in `possible` and makes the set variable part of the environment specified by `env`, where the set consists of enumerated values. By default, its name is the empty string, but you can specify a `name` of your own choice.

`public **IloAnySetVar**(const IloEnv env, const IloAnyArray possible, const IloAnyArray required, const char * name=0)`

With this constructor, you can specify both the `required` and the `possible` sets that characterize the instance of `IloAnySetVar` that it creates. By default, its name is the empty string, but you can specify a `name` of your own choice.

`public **IloAnySetVar**(const IloAnyCollection possible, const char * name=0)`

This constructor creates a constrained set variable and makes it part of the environment `env`, where the set consists of integer values.

`public **IloAnySetVar**(const IloAnyCollection possible, const IloAnyCollection required, const char * name=0)`

This constructor creates a constrained set variable and makes it part of the environment `env`, where the set consists of integer values.

Methods

`public void **addPossible**(IloAny elt) const`

This member function adds `elt` to the set of possible elements of the invoking set variable.

> **Note**
>
> The member function `addPossible` notifies Concert Technology algorithms about this change of this invoking object.

`public void` **addRequired**`(IloAny elt) const`

This member function adds `elt` to the set of required elements of the invoking set variable.

> **Note**
>
> The member function `addRequired` notifies Concert Technology algorithms about this change of this invoking object.

`public IloIntSetVarI *` **getImpl**`() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public void` **getPossibleSet**`(IloAnySet set) const`

This member function accesses the possible elements of the invoking set variable and puts those elements into its argument `set`.

`public IloAnySet` **getPossibleSet**`() const`

This member function returns the possible elements of the invoking set variable.

`public IloAnySet::Iterator` **getPossibleSetIterator**`() const`

This member function returns an `IloAnySet::Iterator` to traverse the possible elements of the invoking set variable.

`public void` **getRequiredSet**`(IloAnySet set) const`

This member function accesses the required elements of the invoking set variable and puts those elements into its argument `set`.

`public IloAnySet` **getRequiredSet**`() const`

This member function returns the required elements of the invoking set variable.

`public IloAnySet::Iterator` **getRequiredSetIterator**`() const`

This member function returns an `IloAnySet::Iterator` to traverse the required elements of the invoking set variable.

`public void` **removePossible**`(IloAny elt) const`

This member function removes `elt` as a possible element of the invoking set variable.

Note

The member function `removePossible` notifies Concert Technology algorithms about this change of this invoking object.

```
public void removeRequired(IloAny elt) const
```

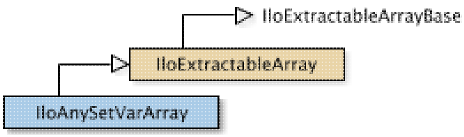
This member function removes `elt` as a required element of the invoking set variable.

Note

The member function `removeRequired` notifies Concert Technology algorithms about this change of this invoking object.

Class IloAnySetVarArray

Definition file: ilconcert/iloanyset.h



For IBM® ILOG® Solver: array class of the set variable class `IloAnySetVar`.
For each basic type, Concert Technology defines a corresponding array class. `IloAnySetVarArray` is the array class of the set variable class for enumerated values (`IloAnySetVar`) in a model.

Instances of `IloAnySetVarArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

For more information on arrays, see the concept Arrays

See Also: IloAnySetVar, IloModel, operator<<

Constructor Summary	
public	`IloAnySetVarArray(IloDefaultArrayI * i=0)`
public	`IloAnySetVarArray(const IloEnv env, IloInt n=0)`

Method Summary	
public void	`add(IloInt more, const IloAnySetVar x)`
public void	`add(const IloAnySetVar x)`
public void	`add(const IloAnySetVarArray array)`
public IloAnySetVar	`operator[](IloInt i) const`
public IloAnySetVar &	`operator[](IloInt i)`

Inherited Methods from `IloExtractableArray`
`add, add, add, endElements, setNames`

Constructors

`public IloAnySetVarArray(IloDefaultArrayI * i=0)`

This constructor creates an empty extensible array of set variables, where each set is a set of enumerated values. You cannot create instances of the undocumented class `IloDefaultArrayI`. As an argument in this default constructor, it allows you to pass 0 (zero) as a value to an optional argument in functions and member functions that accept an array as an argument.

`public IloAnySetVarArray(const IloEnv env, IloInt n=0)`

This constructor creates an extensible array of `n` set variables, where each set is a set of enumerated values. Initially, the `n` elements are empty handles.

Methods

`public void add(IloInt more, const IloAnySetVar x)`

This member function appends `x` to the invoking array multiple times. The argument `more` specifies how many times.

`public void add(const IloAnySetVar x)`

This member function appends `x` to the invoking array.

`public void add(const IloAnySetVarArray array)`

This member function appends the elements in `array` to the invoking array.

`public IloAnySetVar operator[](IloInt i) const`

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`. On `const` arrays, Concert Technology uses the `const` operator:

```
IloAnySetVar operator[] (IloInt i) const;
```

`public IloAnySetVar & operator[](IloInt i)`

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`.

Class IloAnyTernaryPredicate

Definition file: ilconcert/ilotupleset.h

For IBM ILOG Solver: defines ternary predicates on objects in a model.
This class makes it possible for you to define ternary predicates operating on arbitrary objects in a model. A predicate is an object with a member function (such as `IloAnyTernaryPredicate::isTrue`) that checks whether or not a property is satisfied by an ordered set of (pointers to) objects. A ternary predicate checks an ordered set of three objects.

Defining a New Class of Predicates

Predicates, like other Concert Technology objects, depend on two classes: a handle class, `IloAnyTernaryPredicate`, and an implementation class, such as `IloAnyTernaryPredicateI`, where an object of the handle class contains a data member (the handle pointer) that points to an object (its implementation object) of an instance of `IloAnyTernaryPredicateI` allocated in a Concert Technology environment. As a Concert Technology user, you will be working primarily with handles.

If you define a new class of predicates yourself, you must define its implementation class together with the corresponding virtual member function `isTrue`, as well as a member function that returns an instance of the handle class `IloAnyTernaryPredicate`.

Arity

As a developer, you can use predicates in Concert Technology applications to define your own constraints that have not already been predefined in Concert Technology. In that case, the *arity* of the predicate (that is, the number of constrained variables involved in the predicate, and thus the size of the array that the member function `IloAnyTernaryPredicate::isTrue` must check) must be three.

See Also: IloTableConstraint

Constructor and Destructor Summary	
public	`IloAnyTernaryPredicate()`
public	`IloAnyTernaryPredicate(IloAnyTernaryPredicateI * impl)`

Method Summary	
public IloAnyTernaryPredicateI *	`getImpl() const`
public IloBool	`isTrue(const IloAny val1, const IloAny val2, const IloAny val3)`
public void	`operator=(const IloAnyTernaryPredicate & h)`

Constructors and Destructors

public **IloAnyTernaryPredicate**()

This constructor creates an empty ternary predicate. In other words, the predicate is an empty handle with a null handle pointer. You must assign the elements of the predicate before you attempt to access it, just as you would any other pointer. Any attempt to access it before this assignment will throw an exception (an instance of `IloSolver::SolverErrorException`).

public **IloAnyTernaryPredicate**(IloAnyTernaryPredicateI * impl)

This constructor creates a handle object (an instance of the class `IloAnyTernaryPredicate`) from a pointer to an implementation object (an instance of the implementation class `IloAnyTernaryPredicateI`).

Methods

`public IloAnyTernaryPredicateI * **getImpl**() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public IloBool **isTrue**(const IloAny val1, const IloAny val2, const IloAny val3)`

This member function returns `IloTrue` if the values `val1`, `val2`, and `val3` make the invoking ternary predicate valid. It returns `IloFalse` otherwise.

`public void **operator=**(const IloAnyTernaryPredicate & h)`

This assignment operator copies `h` into the invoking predicate by assigning an address to the handle pointer of the invoking object. That address is the location of the implementation object of the argument `h`. After execution of this operator, both the invoking predicate and `h` point to the same implementation object.

Class IloAnyTupleSet

Definition file: ilconcert/ilotupleset.h

Ordered set of values as an array.
A tuple is an ordered set of values represented by an array. A *set* of enumerated tuples in a model is represented by an instance of `IloAnyTupleSet`. That is, the elements of a tuple *set* are tuples of enumerated values (such as pointers). The number of values in a tuple is known as the *arity* of the tuple, and the arity of the tuples in a set is called the *arity* of the set. (In contrast, the number of tuples in the set is known as the *cardinality* of the set.)

As a handle class, `IloAnyTupleSet` manages certain set operations efficiently. In particular, elements can be added to such a set. It is also possible to search a given set with the member function `IloAnyTupleSet::isIn` to see whether or not the set contains a given element.

In addition, a set of tuples can represent a constraint defined on a constrained variable, either as the set of *allowed* combinations of values of the constrained variable on which the constraint is defined, or as the set of *forbidden* combinations of values.

There are a few conventions governing tuple sets:

- When you create the set, you must specify the arity of the tuple-elements it contains.
- You use the member function `IloAnyTupleSet::add` to add tuples to the set. You can add tuples to the set in a model; you cannot add tuples to an instance of this class during a search, nor inside a constraint, nor inside a goal.

Concert Technology will throw an exception (an instance of `IloSolver::SolverErrorException`) if you attempt:

- to add a tuple with a different number of variables from the arity of the set;
- to search for a tuple with an arity different from the set arity.

You do not have to worry about memory allocation. If you respect these conventions, Concert Technology manages allocation and de-allocation transparently for you.

See Also the class `IlcIntTupleSet` in the *IBM ILOG CP Optimizer Reference Manual* and the *ILOG Solver Reference Manual*.

See Also: IloAnyTupleSetIterator, IloTableConstraint, IloExtractable

Constructor Summary	
public	`IloAnyTupleSet(const IloEnv env, const IloInt arity)`

Method Summary	
public IloBool	`add(const IloAnyArray tuple) const`
public IloInt	`getArity() const`
public IloAnyTupleSetI *	`getImpl() const`
public IloBool	`isIn(const IloAnyArray tuple) const`
public IloBool	`remove(const IloAnyArray tuple) const`

Constructors

`public IloAnyTupleSet(const IloEnv env, const IloInt arity)`

This constructor creates a set of tuples (an instance of the class `IloAnyTupleSet`) with the arity specified by `arity`.

Methods

`public IloBool add(const IloAnyArray tuple) const`

This member function adds a tuple represented by the array `tuple` to the invoking set. If you attempt to add an element that is already in the set, that element will *not* be added again. Added elements are not copied; that is, there is no memory duplication. Concert Technology will throw an exception if the size of the array is not equal to the arity of the invoking set. You may use this member function to add tuples to the invoking set in a model; you may not add tuples in this way during a search, inside a constraint, or inside a goal. For those purposes, see `IlcIntTupleSet`, documented in the *IBM ILOG CP Optimizer Reference Manual* and the *IBM ILOG Solver Reference Manual*.

`public IloInt getArity() const`

This member function returns the arity of the tupleset.

`public IloAnyTupleSetI * getImpl() const`

This member function returns a pointer to the implementation object of the invoking extractable object. This member function is useful when you need to be sure that you are using the same copy of the invoking extractable object in more than one situation.

`public IloBool isIn(const IloAnyArray tuple) const`

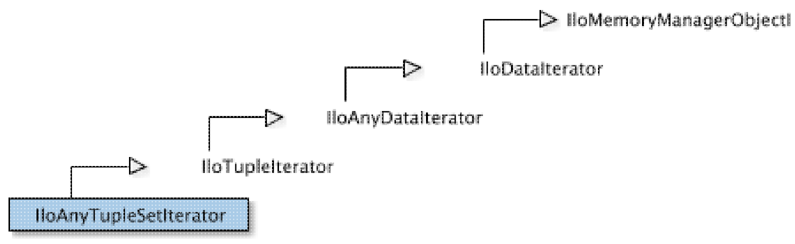
This member function returns `IloTrue` if `tuple` belongs to the invoking set. Otherwise, it returns `IloFalse`. Concert Technology will throw an exception if the size of the array is not equal to the arity of the invoking set.

`public IloBool remove(const IloAnyArray tuple) const`

This member function removes `tuple` from the invoking set in a model. You may use this member function to remove tuples from the invoking set in a model; you may not remove tuples in this way during a search, inside a constraint, or inside a goal. For those purposes, see `IlcIntTupleSet` documented in the *IBM ILOG CP Optimizer Reference Manual* and the *IBM ILOG Solver Reference Manual*.

Class IloAnyTupleSetIterator

Definition file: ilconcert/ilotupleset.h



Iterator to traverse enumerated values of a tuple-set.
An instance of the class `IloAnyTupleSetIterator` is an iterator that traverses the elements of a finite set of tuples of enumerated values (instance of `IloAnyTupleSet`).

See Also the class `IlcAnyTupleSet` in the *IBM ILOG Solver Reference Manual*.

Constructor Summary	
`public`	`IloAnyTupleSetIterator(const IloEnv env, IloAnyTupleSet tset)`

Method Summary	
`public IloAnyArray`	`operator*() const`

Constructors

`public` **`IloAnyTupleSetIterator`**`(const IloEnv env, IloAnyTupleSet tset)`

This constructor creates an iterator associated with `tSet` to traverse its elements.

Methods

`public IloAnyArray` **`operator*`**`() const`

This operator returns the current element, the one to which the invoking iterator points.

Class IloAnyValueSelector

Definition file: ilsolver/ilosolverany.h
Include file: <ilsolver/ilosolver.h>

Solver lets you create value selectors to control the order in which the values in the domain of a constrained enumerated variable are tried during the search for a solution.

The class `IloAnyValueSelector` represents value selectors in a Concert Technology *model*. The class `IlcAnySelect` represents value selectors internally in a Solver search.

This class is the handle class of the modeling object that wraps the search object. When search starts, `IloAnyValueSelectorI` is extracted into an instance of `IlcAnySelectI`.

See Also: IlcAnySelect, IloAnyValueSelectorI

Constructor Summary	
`public`	`IloAnyValueSelector()`
`public`	`IloAnyValueSelector(IloAnyValueSelectorI * impl)`

Method Summary	
`public IloAnyValueSelectorI *`	`getImpl() const`
`public void`	`operator=(const IloAnyValueSelector & h)`

Constructors

`public` **`IloAnyValueSelector`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IloAnyValueSelector`**`(IloAnyValueSelectorI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

Methods

`public IloAnyValueSelectorI *` **`getImpl`**`() const`

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

`public void` **`operator=`**`(const IloAnyValueSelector & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IloAnyValueSelectorI

Definition file: ilsolver/ilosolverany.h
Include file: <ilsolver/ilosolver.h>

This class is the implementation class for `IloAnyValueSelector`.

The class `IloAnyValueSelectorI` is the implementation class for value selectors in a Concert Technology *model*. The class `IlcAnySelectI` is the implementation class for value selectors internally in a Solver search.

This class is the modeling object that wraps the search object. When search starts, `IloAnyValueSelectorI` is extracted into an instance of `IlcAnySelectI`.

To define new selection criteria, you define both a subclass of `IloAnyValueSelectorI` and a subclass of `IlcAnySelectI`.

See Also: IlcAnySelect, IloAnyValueSelector

Constructor and Destructor Summary	
`public`	`IloAnyValueSelectorI(IloEnvI *)`
`public`	`~IloAnyValueSelectorI()`

Method Summary	
`public virtual void`	`display(ostream &) const`
`public virtual IlcAnySelect`	`extract(const IloSolver solver) const`
`public IloEnvI *`	`getEnv() const`
`public virtual IloAnyValueSelectorI *`	`makeClone(IloEnvI * env) const`

Constructors and Destructors

`public` **`IloAnyValueSelectorI`**`(IloEnvI *)`

This constructor creates an instance of the class `IloAnyValueSelectorI`. This constructor should not be called directly as this class is an abstract class. This constructor is called automatically in the constructor of its subclasses.

`public` **`~IloAnyValueSelectorI`**`()`

This destructor is called automatically by the destructor of its subclasses. It frees memory used by the invoking object.

Methods

`public virtual void` **`display`**`(ostream &) const`

This member function prints the invoking value selector on an output stream.

```
public virtual IlcAnySelect extract(const IloSolver solver) const
```

In general terms, in Concert Technology, the objects of a model must be extracted for an algorithm (an instance of one of the subclasses of `IloAlgorithm`, such as `IloSolver`). This member function returns the internal value selector extracted for `solver` from the invoking value selector of a model.

```
public IloEnvI * getEnv() const
```

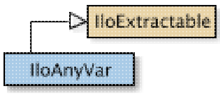
This member function returns the environment to which the invoking value selector belongs. A value selector belongs to exactly one environment; different environments cannot share the same value selector.

```
public virtual IloAnyValueSelectorI * makeClone(IloEnvI * env) const
```

This member function is called internally to duplicate the current value selector.

Class IloAnyVar

Definition file: ilconcert/iloany.h



For IBM® ILOG® Solver: a class to represent an enumerated variable.
An instance of this class offers a convenient way to represent an enumerated variable in Concert Technology.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloAny, IloAnyVarArray, IloExtractable, IloModel

Constructor Summary	
public	IloAnyVar()
public	IloAnyVar(IloNumVarI * impl)
public	IloAnyVar(const IloEnv env, const IloAnyArray array, const char * name=0)

Method Summary	
public IloNumVarI *	getImpl() const
public void	getPossibleValues(IloAnyArray values) const
public void	setPossibleValues(const IloAnyArray values)

Inherited Methods from `IloExtractable`
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Constructors

`public IloAnyVar()`

This constructor creates an empty handle. You must initialize it before you use it.

`public IloAnyVar(IloNumVarI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public IloAnyVar(const IloEnv env, const IloAnyArray array, const char * name=0)`

This constructor creates a constrained enumerated variable from the values in `array` and makes the variable part of the environment specified by `env`. By default, its name is the empty string, but you can specify a `name` of your own choice.

Methods

`public IloNumVarI * getImpl() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public void **getPossibleValues**(IloAnyArray values) const`

This constructor creates a constrained enumerated variable from the given collection.

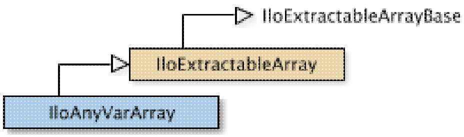
This member function accesses the possible values of the invoking enumerated variable and puts those values into its argument `values`.

`public void **setPossibleValues**(const IloAnyArray values)`

This member function sets `values` as the domain of the invoking enumerated variable.

Class IloAnyVarArray

Definition file: ilconcert/iloany.h



For IBM® ILOG® Solver: a class to represent an array of enumerated variables.
For each basic type, Concert Technology defines a corresponding array class. `IloAnyVarArray` is the array class of the enumerated variable class (`IloAnyVar`) for a model. The parent class for `IloAnyVarArray` is the class `IloExtractableArray`.

Instances of `IloAnyVarArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloAnyVar, IloModel, operator<<

Constructor Summary	
public	IloAnyVarArray(IloDefaultArrayI * i=0)
public	IloAnyVarArray(const IloEnv env, IloInt n=0)

Method Summary	
public void	add(IloInt more, const IloAnyVar x)
public void	add(const IloAnyVar x)
public void	add(const IloAnyVarArray array)
public IloAnyVar	operator[](IloInt i) const
public IloAnyVar &	operator[](IloInt i)

Inherited Methods from `IloExtractableArray`
add, add, add, endElements, setNames

Constructors

`public` **`IloAnyVarArray`**`(IloDefaultArrayI * i=0)`

This constructor creates an empty extensible array of enumerated variables. You cannot create instances of the undocumented class `IloDefaultArrayI`. As an argument in this default constructor, it allows you to pass 0 (zero) as a value to an optional argument in functions and member functions that accept an array as an argument.

`public` **`IloAnyVarArray`**`(const IloEnv env, IloInt n=0)`

This constructor creates an extensible array of `n` enumerated variables. Initially, the `n` elements are empty handles.

Methods

```
public void add(IloInt more, const IloAnyVar x)
```

This member function appends `x` to the invoking array multiple times. The argument `more` specifies how many times.

```
public void add(const IloAnyVar x)
```

This member function appends `x` to the invoking array.

```
public void add(const IloAnyVarArray array)
```

This member function appends the elements in `array` to the invoking array.

```
public IloAnyVar operator[](IloInt i) const
```

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`. On `const` arrays, Concert Technology uses the `const` operator:

```
IloAnyVar operator[] (IloInt i) const;
```

```
public IloAnyVar & operator[](IloInt i)
```

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`.

Class IloArray<>

Definition file: ilconcert/iloenv.h

A template to create classes of arrays for elements of a given class.
This C++ template creates a class of arrays for elements of a given class. In other words, you can use this template to create arrays of Concert Technology objects; you can also use this template to create arrays of arrays (that is, multidimensional arrays).

In its synopsis, `X` represents a class, `x` is an instance of the class `X`. This template creates the array class (`IloArrayX`) for any class in Concert Technology, including classes with names in the form `IloXArray`, such as `IloExtractableArray`. Concert Technology predefines the array classes listed here as **See Also**. The member functions defined by this template are documented in each of those predefined classes.

The classes you create in this way consist of extensible arrays. That is, you can add elements to the array as needed.

Deleting Arrays

The member function `end` created by this template deletes only the array; the member function does not delete the elements of the array.

Copying Arrays

Like certain other Concert Technology classes, a class of arrays created by `IloArray` is a handle class corresponding to an implementation class. In other words, an instance of an `IloArray` class is a handle pointing to a corresponding implementation object. More than one handle may point to the same implementation object.

Input and Output of Multidimensional Arrays

The template `operator >>` makes it possible to read numeric values from a file in the format `[x, y, z, ...]` where `x`, `y`, `z` are the results of the `operator >>` for class `X`. Class `X` must provide a default constructor for `operator >>` to work. That is, the statement `X x;` must work for `X`. This input operator is limited to numeric values.

Likewise, the template `operator <<` makes it possible to write to a file in the format `[x, y, z, ...]` where `x`, `y`, `z` are the results of the `operator <<` for class `X`. (This output operator is *not* limited to numeric values, as the input operator is.)

These two operators make it possible to read and write multidimensional arrays of numeric values like this:

```
IloArray<IloArray<IloIntArray> >
```

(Notice the space between > > at the end of that statement. It is necessary in C++.)

However, there is a practical limit of four on the number of dimensions supported by the input operator for reading multidimensional arrays. This limit is due to the inability of certain C++ compilers to support templates correctly. Specifically, you can read input by means of the input operator for multidimensional arrays of one, two, three, or four dimensions. There is no such limit on the number of dimensions with respect to the output operator for multidimensional arrays.

See Also these classes in the *IBM ILOG CPLEX Reference Manual*: `IloSemiContVarArray`, `IloSOS1Array`, `IloSOS2Array`, `IloNumColumnArray`.

See Also these classes in the *IBM ILOG Solver Reference Manual*: `IloAnyArray`, `IloAnySetVarArray`, `IloAnyVarArray`, `IloFloatArray`, `IloFloatVarArray`.

See Also: IloBoolArray, IloBoolVarArray, IloConstraintArray, IloExprArray, IloExtractableArray, IloIntArray, IloIntVarArray, IloNumVarArray, IloRangeArray, IloSolutionArray

Constructor Summary	
public	`IloArray(IloEnv env, IloInt max=0)`

Method Summary	
public void	`add(IloArray< X > ax) const`
public void	`add(IloInt more, X x) const`
public void	`add(X x) const`
public void	`clear()`
public void	`end()`
public IloEnv	`getEnv() const`
public IloInt	`getSize() const`
public X &	`operator[](IloInt i)`
public const X &	`operator[](IloInt i) const`
public void	`remove(IloInt first, IloInt nb=1)`

Constructors

`public` **`IloArray`**`(IloEnv env, IloInt max=0)`

This constructor creates an array of `max` elements, all of which are empty handles.

Methods

`public void` **`add`**`(IloArray< X > ax) const`

This member function appends the elements in `ax` to the invoking array.

`public void` **`add`**`(IloInt more, X x) const`

This member function appends `x` to the invoking array multiple times. The argument `more` specifies how many times.

`public void` **`add`**`(X x) const`

This member function appends `x` to the invoking array.

`public void` **`clear`**`()`

This member function removes all the elements from the invoking array. In other words, it produces an empty array.

`public void` **`end`**`()`

This member function first removes the invoking extractable object from all other extractable objects where it is used (such as a model, ranges, etc.) and then deletes the invoking extractable object. That is, it frees all the resources used by the invoking object. After a call to this member function, you cannot use the invoking extractable object again.

`public IloEnv` **`getEnv`**`() const`

This member function returns the environment where the invoking array was created. The elements of the invoking array belong to the same environment.

`public IloInt` **`getSize`**`() const`

This member function returns an integer specifying the size of the invoking array. An empty array has size 0 (zero).

`public X &` **`operator[]`**`(IloInt i)`

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`.

`public const X &` **`operator[]`**`(IloInt i) const`

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`. On `const` arrays, Concert Technology uses the `const` operator:

```
IloArray operator[] (IloInt i) const;
```

`public void` **`remove`**`(IloInt first, IloInt nb=1)`

This member function removes elements from the invoking array. It begins removing elements at the index specified by `first`, and it removes `nb` elements (`nb = 1` by default).

Class IloBarrier

Definition file: ilconcert/ilothread.h

A system class to synchronize threads at a specified number.
The class `IloBarrier` provides synchronization primitives adapted to Concert Technology. A barrier, an instance of this class, serves as a rendezvous for a specific number of threads. After you create a barrier for n threads, the first n-1 threads to reach that barrier will be blocked. The nth thread to arrive at the barrier completes the synchronization and wakes up the n-1 threads already waiting at that barrier. When the nth thread arrives, the barrier resets itself. Any other thread that arrives at this point is blocked and will participate in a new barrier of size n.

> **Note**
>
> The class `IloBarrier` has nothing to do with the IBM ILOG CPLEX barrier optimizer.

System Class

`IloBarrier` is a system class.

Most Concert Technology classes are actually handle classes whose instances point to objects of a corresponding implementation class. For example, instances of the Concert Technology class `IloNumVar` are handles pointing to instances of the implementation class `IloNumVarI`. Their allocation and de-allocation in a Concert Technology environment are managed by an instance of `IloEnv`.

However, system classes, such as `IloBarrier`, differ from that Concert Technology pattern. `IloBarrier` is an ordinary C++ class. Its instances are allocated on the C++ heap.

Instances of `IloBarrier` are not automatically de-allocated by a call to `IloEnv::end`. You must explicitly destroy instances of `IloBarrier` by means of a call to the delete operator (which calls the appropriate destructor) when your application no longer needs instances of this class.

Furthermore, you should not allocate—neither directly nor indirectly—any instance of `IloBarrier` in a Concert Technology environment because the destructor for that instance of `IloBarrier` will never be called automatically by `IloEnv::end` when it cleans up other Concert Technology objects in that Concert Technology environment.

For example, it is not a good idea to make an instance of `IloBarrier` part of a conventional Concert Technology model allocated in a Concert Technology environment because that instance will not automatically be de-allocated from the Concert Technology environment along with the other Concert Technology objects.

De-allocating Instances of IloBarrier

Instances of `IloBarrier` differ from the usual Concert Technology objects because they are not allocated in a Concert Technology environment, and their de-allocation is not managed automatically for you by `IloEnv::end`. Instead, you must explicitly destroy instances of `IloBarrier` by calling the delete operator when your application no longer needs those objects.

See Also: IloCondition, IloFastMutex

Constructor Summary	
`public`	`IloBarrier(int count)`

Method Summary

public int	wait()

Constructors

`public IloBarrier(int count)`

This constructor creates an instance of `IloBarrier` of size `count` and allocates it on the C++ heap (not in a Concert Technology environment).

Methods

`public int wait()`

The first `count-1` calls to this member function block the calling thread. The last call (that is, the call numbered `count`) wakes up all the `count-1` waiting threads. Once a thread has been woken up, it leaves the barrier. When a thread leaves the barrier (that is, when it returns from the `wait` call), it will return either 1 (one) or 0 (zero). If the thread returns 0, the barrier is not yet empty. If the thread returns 1, it was the last thread at the barrier.

A nonempty barrier contains blocked threads or exiting threads.

Class IloBaseEnvMutex

Definition file: ilconcert/iloenv.h

A class to initialize multithreading in an application.
An instance of this base class in the function `IloInitMT` initializes multithreading in a Concert Technology application. For a general purpose mutex, see the class `IloFastMutex`.

See Also: IloFastMutex, IloInitMT

Method Summary	
`public virtual void`	`lock()`
`public virtual void`	`unlock()`

Methods

`public virtual void` **`lock`**`()`

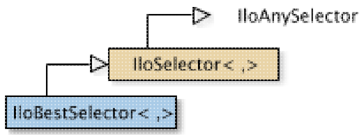
This member function locks a mutex.

`public virtual void` **`unlock`**`()`

This member function unlocks a mutex.

Class IloBestSelector<,>

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>



The IloBestSelector template class is a subclass of selector that implements the selection of an object of type IloObject provided by a container of typeIloContainer based on:

- A visitor of type IloVisitor<IloObject,IloContainer>
- A predicate of type IloPredicate<IloObject>
- A comparator of type IloComparator<IloObject>

The selector will select an instance of IloObject visited by the visitor that satisfies the predicate and that is the best according to the comparator.

In the case where no visitor is provided, the selector uses the default visitor provided for the template <IloObject,IloContainer> (see macro ILODEFAULTVISITOR). In the case where no predicate is provided, the default is a predicate that always returns IloTrue. In the case where no comparator is provided, the default is a comparator that considers all objects as being equal.

For more information, see Selectors.

See Also: IloVisitor, IloPredicate, IloComparator

Constructor Summary	
public	IloBestSelector(IloMemoryManager manager) Creates a selector returning an IloObject from an IloContainer.
public	IloBestSelector(IloVisitor< IloObject, IloContainer > visitor, IloPredicate< IloObject > pred=0, IloComparator< IloObject > cmp=0) Creates a selector returning an IloObject from an IloContainer.
public	IloBestSelector(IloPredicate< IloObject > pred, IloComparator< IloObject > cmp=0) Creates a selector returning an IloObject from an IloContainer.
public	IloBestSelector(IloComparator< IloObject > cmp) Creates a selector returning an IloObject from an IloContainer.
public	IloBestSelector(IloVisitor< IloObject, IloContainer > visitor, IloComparator< IloObject > cmp) Creates a selector returning an IloObject from an IloContainer.

Method Summary	
public IloComparator< IloObject >	getComparator() const Returns the selector's comparator.
public IloPredicate< IloObject >	getPredicate() const

	Returns the selector's predicate.
public IloVisitor< IloObject, IloContainer >	getVisitor() const Returns the selector's visitor.

Inherited Methods from `IloSelector`
select

Constructors

`public IloBestSelector(IloMemoryManager manager)`

Creates a selector returning an `IloObject` from an `IloContainer`.

internal

This constructor creates a selector that uses a default visitor allocated on the memory mananger `manager`.

`public IloBestSelector(IloVisitor< IloObject, IloContainer > visitor, IloPredicate< IloObject > pred=0, IloComparator< IloObject > cmp=0)`

Creates a selector returning an `IloObject` from an `IloContainer`.

This constructor creates a selector that uses the visitor `visitor`, the predicate `pred`, and the comparator `cmp` given as arguments.

`public IloBestSelector(IloPredicate< IloObject > pred, IloComparator< IloObject > cmp=0)`

Creates a selector returning an `IloObject` from an `IloContainer`.

This constructor creates a selector that uses the predicate `pred` and the comparator `cmp` given as arguments.

`public IloBestSelector(IloComparator< IloObject > cmp)`

Creates a selector returning an `IloObject` from an `IloContainer`.

This constructor creates a selector that uses the comparator `cmp` given as argument.

`public IloBestSelector(IloVisitor< IloObject, IloContainer > visitor, IloComparator< IloObject > cmp)`

Creates a selector returning an `IloObject` from an `IloContainer`.

This constructor creates a selector that uses the visitor `visitor` and the comparator `cmp` given as arguments.

Methods

`public IloComparator< IloObject > getComparator() const`

Returns the selector's comparator.

This member function returns the comparator associated with the invoking selector.

```
public IloPredicate< IloObject > getPredicate() const
```

Returns the selector's predicate.

This member function returns the predicate associated with the invoking selector.

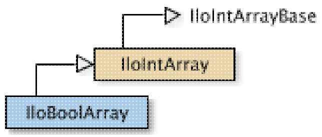
```
public IloVisitor< IloObject, IloContainer > getVisitor() const
```

Returns the selector's visitor.

This member function returns the visitor associated with the invoking selector.

Class IloBoolArray

Definition file: ilconcert/iloenv.h



The array class of the basic Boolean class for a model.
`IloBoolArray` is the array class of the basic Boolean class for a model. It is a handle class. The implementation class for `IloBoolArray` is the undocumented class `IloBoolArrayI`.

Instances of `IloBoolArray` are extensible. (They differ from instances of `IlcBoolArray` in this respect.) References to an array change whenever an element is added to or removed from the array.

For each basic type, Concert Technology defines a corresponding array class. That array class is a handle class. In other words, an object of that class contains a pointer to another object allocated in a Concert Technology environment associated with a model. Exploiting handles in this way greatly simplifies the programming interface since the handle can then be an automatic object: as a developer using handles, you do not have to worry about memory allocation.

As handles, these objects should be passed by value, and they should be created as automatic objects, where "automatic" has the usual C++ meaning.

Member functions of a handle class correspond to member functions of the same name in the implementation class.

Assert and NDEBUG

Most member functions of the class `IloBoolArray` are inline functions that contain an `assert` statement. This statement checks that the handle pointer is not null. These statements can be suppressed by the macro `NDEBUG`. This option usually reduces execution time. The price you pay for this choice is that attempts to access through null pointers are not trapped and usually result in memory faults.

See Also: IloBool

Constructor Summary	
`public`	`IloBoolArray(IloArrayI * i=0)`
`public`	`IloBoolArray(const IloEnv env, IloInt n=0)`
`public`	`IloBoolArray(const IloEnv env, IloInt n, const IloBool v0, const IloBool v1...)`

Method Summary	
`public void`	`add(IloInt more, const IloBool x)`
`public void`	`add(const IloBool x)`
`public void`	`add(const IloBoolArray x)`

Inherited Methods from `IloIntArray`
`contains, contains, discard, discard, operator[], operator[], operator[], toNumArray`

Constructors

`public IloBoolArray(IloArrayI * i=0)`

This constructor creates an array of Boolean values from an implementation object.

`public IloBoolArray(const IloEnv env, IloInt n=0)`

This constructor creates an array of `n` Boolean values for use in a model in the environment specified by `env`. By default, its elements are empty handles.

`public IloBoolArray(const IloEnv env, IloInt n, const IloBool v0, const IloBool v1...)`

This constructor creates an array of `n` Boolean values; the elements of the new array take the corresponding values: `v0, v1, ...,v(n-1).`

Methods

`public void add(IloInt more, const IloBool x)`

This member function appends `x` to the invoking array of Boolean values; it appends `x more` times.

`public void add(const IloBool x)`

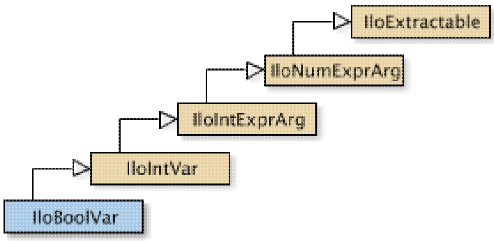
This member function appends the value `x` to the invoking array.

`public void add(const IloBoolArray x)`

This member function appends the values in the array `x` to the invoking array.

Class IloBoolVar

Definition file: ilconcert/iloexpression.h



An instance of this class represents a constrained Boolean variable in a Concert Technology model. Boolean variables are also known as binary decision variables. They can assume the values 0 (zero) or 1 (one).

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

What Is Extracted

An instance of `IloBoolVar` is extracted by `IloSolver` (documented in the *IBM ILOG Solver Reference Manual*) as an instance of the class `IlcBoolVar` (also documented in the *IBM ILOG Solver Reference Manual).*

An instance of `IloBoolVar` is extracted by `IloCplex` (documented in the *IBM ILOG CPLEX Reference Manual)* as a column representing a numeric variable of type `Bool` with bounds as specified by `IloBoolVar`.

See Also: IloIntVar, IloNumVar

Constructor Summary	
`public`	`IloBoolVar(IloEnv env, IloInt min=0, IloInt max=1, const char * name=0)`
`public`	`IloBoolVar(IloEnv env, const char * name)`
`public`	`IloBoolVar(const IloAddNumVar & column, const char * name=0)`

Inherited Methods from `IloIntVar`
`getImpl, getLB, getMax, getMin, getUB, setBounds, setLB, setMax, setMin, setPossibleValues, setUB`

Inherited Methods from `IloIntExprArg`
`getImpl`

Inherited Methods from `IloNumExprArg`
`getImpl`

Inherited Methods from `IloExtractable`
`asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject`

Constructors

`public IloBoolVar(IloEnv env, IloInt min=0, IloInt max=1, const char * name=0)`

This constructor creates a Boolean variable and makes it part of the environment `env`. By default, the Boolean variable assumes a value of 0 (zero) or 1 (one). By default, its name is the empty string, but you can specify a name of your own choice.

`public IloBoolVar(IloEnv env, const char * name)`

This constructor creates a Boolean variable and makes it part of the environment `env`. By default, its name is the empty string, but you can specify a name of your own choice.

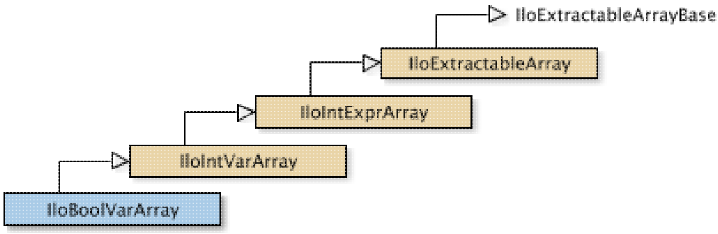
`public IloBoolVar(const IloAddNumVar & column, const char * name=0)`

This constructor creates an instance of `IloBoolVar` like this:

```
IloNumVar(column, 0.0, 1.0, ILOBOOL, name);
```

Class IloBoolVarArray

Definition file: ilconcert/iloexpression.h



The array class of the Boolean variable class.
For each basic type, Concert Technology defines a corresponding array class. `IloBoolVarArray` is the array class of the Boolean variable class for a model. It is a handle class.

Instances of `IloBoolVarArray` are extensible.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloBoolVar

Constructor Summary	
`public`	`IloBoolVarArray(IloDefaultArrayI * i=0)`
`public`	`IloBoolVarArray(const IloEnv env, IloInt n)`
`public`	`IloBoolVarArray(const IloEnv env, const IloNumColumnArray columnarray)`

Method Summary	
`public void`	`add(IloInt more, const IloBoolVar x)`
`public void`	`add(const IloBoolVar x)`
`public void`	`add(const IloBoolVarArray x)`
`public IloBoolVar`	`operator[](IloInt i) const`
`public IloBoolVar &`	`operator[](IloInt i)`
`public IloIntExprArg`	`operator[](IloIntExprArg anIntegerExpr) const`

Inherited Methods from **`IloIntVarArray`**
`add, add, add, endElements, operator[], operator[], operator[], toNumVarArray`

Inherited Methods from **`IloIntExprArray`**
`add, add, add, endElements, operator[], operator[], operator[]`

Inherited Methods from **`IloExtractableArray`**
`add, add, add, endElements, setNames`

Constructors

`public` **`IloBoolVarArray`**`(IloDefaultArrayI * i=0)`

This constructor creates an empty extensible array of Boolean variables.

`public **IloBoolVarArray**(const IloEnv env, IloInt n)`

This constructor creates an extensible array of `n` Boolean variables.

`public **IloBoolVarArray**(const IloEnv env, const IloNumColumnArray columnarray)`

This constructor creates an extensible array of Boolean variables from a column array.

Methods

`public void **add**(IloInt more, const IloBoolVar x)`

This member function appends `x` to the invoking array of Boolean variables. The argument `more` specifies how many times.

`public void **add**(const IloBoolVar x)`

This member function appends the value `x` to the invoking array.

`public void **add**(const IloBoolVarArray x)`

This member function appends the variables in the array `x` to the invoking array.

`public IloBoolVar **operator[]**(IloInt i) const`

This operator returns a reference to the extractable object located in the invoking array at the position specified by the index `i`. On `const` arrays, Concert Technology uses the `const` operator:

```
IloBoolVar operator[] (IloInt i) const;
```

`public IloBoolVar & **operator[]**(IloInt i)`

This operator returns a reference to the extractable object located in the invoking array at the position specified by the index `i`.

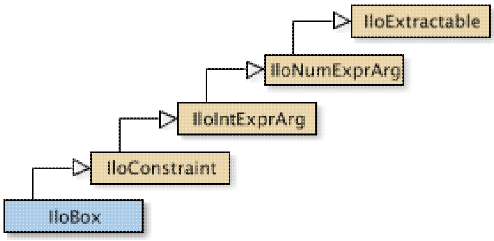
`public IloIntExprArg **operator[]**(IloIntExprArg anIntegerExpr) const`

This subscripting operator returns an expression argument for use in a constraint or expression. For clarity, let's call `A` the invoking array. When `anIntegerExpr` is bound to the value `i`, the domain of the expression is the domain of `A[i]`. More generally, the domain of the expression is the union of the domains of the expressions `A[i]` where the `i` are in the domain of `anIntegerExpr`.

This operator is also known as an element expression.

Class IloBox

Definition file: ilconcert/ilobox.h



For IBM ILOG Solver: multidimensional boxes for multidimensional placement problems.
Instances of the class `IloBox` are multidimensional boxes that appear in multidimensional placement problems. To solve packing or placement problems, you may need to be able to place boxes within a given container. In such a situation, both the boxes to place and the container to hold them are instances of the class `IloBox`.

To specify the containment relation that a given container holds a given box, use the member function `IloBox::contains`.

See Also the class `IlcBox` in the *IBM ILOG Solver Reference Manual* and the class `IlcFilterLevelConstraint` in the *IBM ILOG Solver Reference Manual*.

Constructor Summary	
public	`IloBox()`
public	`IloBox(IloBoxI * impl)`
public	`IloBox(const IloEnv env, IloInt dimensions, const IloIntVarArray origin, const IloIntArray size)`

Method Summary	
public IloInt	`getDimensions()`
public IloBoxI *	`getImpl() const`
public IloIntVar	`getOrigin(IloInt dimension)`
public IloInt	`getSize(IloInt dimension)`

Inherited Methods from `IloConstraint`
`getImpl`

Inherited Methods from `IloIntExprArg`
`getImpl`

Inherited Methods from `IloNumExprArg`
`getImpl`

Inherited Methods from `IloExtractable`
`asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject`

Constructors

`public IloBox()`

This constructor creates an empty handle. You must initialize it before you use it.

`public IloBox(IloBoxI * impl)`

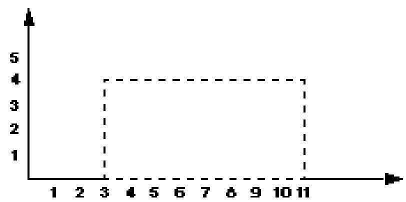
This constructor creates a handle object from a pointer to an implementation object.

`public IloBox(const IloEnv env, IloInt dimensions, const IloIntVarArray origin, const IloIntArray size)`

This constructor creates a box according to the specifications passed in the arguments. The argument `dimensions` specifies the number of dimensions the box has. The arrays `origin` and `size` must contain the same number of elements as the number of dimensions of the box. In dimension *i*, the box extends from `origin[i]` to `origin[i] + size[i]`. For example, the statement

```
IloBox(env, 2, IloIntVarArray (env, 2, 3, 0), IloIntArray(env, 2, 8, 4));
```

creates a box as shown in this illustration.



Methods

`public IloInt getDimensions()`

This member function returns the number of dimensions of the invoking box.

`public IloBoxI * getImpl() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public IloIntVar getOrigin(IloInt dimension)`

This member function returns the origin of the invoking box along the dimension specified by `dimension`.

`public IloInt getSize(IloInt dimension)`

This member function returns the length of the invoking box along the dimension specified by `dimension`.

Class IloBranchSelector

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

An instance of this class represents a branch selector in a Concert Technology model. Branch selectors are useful in *goals* (such as the goal returned by `IloApply` or other instances of `IloGoal`) to shape the exploration of the search tree during the search for a solution.

Constructor Summary	
`public`	`IloBranchSelector()`
`public`	`IloBranchSelector(IloBranchSelectorI * impl)`

Method Summary	
`public IloBranchSelectorI *`	`getImpl() const`
`public void`	`operator=(const IloBranchSelector & h)`

Constructors

`public` **`IloBranchSelector`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IloBranchSelector`**`(IloBranchSelectorI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

Methods

`public IloBranchSelectorI *` **`getImpl`**`() const`

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

`public void` **`operator=`**`(const IloBranchSelector & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IloBranchSelectorI

Definition file: ilsolver/ilosolverint.h

The class `IloBranchSelector` represents branch selectors in a Concert Technology model. The class `IlcBranchSelector` represents branch selectors internally in a Solver search.

A branch selector is used to shape the explored part of the search tree.

This class is the implementation class for `IloBranchSelector`.

Constructor and Destructor Summary	
public	~IloBranchSelectorI()

Method Summary	
public virtual void	display(ostream &) const
public virtual IlcBranchSelector	extract(const IloSolver solver) const
public virtual IloBranchSelectorI *	makeClone(IloEnvI * env) const

Constructors and Destructors

`public` **`~IloBranchSelectorI`**`()`

This destructor is called automatically by the destructor of its subclasses. It frees memory used by the invoking object.

Methods

`public virtual void` **`display`**`(ostream &) const`

This member function prints the invoking branch selector on an output stream.

`public virtual IlcBranchSelector` **`extract`**`(const IloSolver solver) const`

In general terms, in Concert Technology, the objects of a model must be extracted for an algorithm (an instance of one of the subclasses of `IloAlgorithm`, such as `IloSolver`). This member function returns the internal branch selector extracted for `solver` from the invoking branch selector of a model.

`public virtual IloBranchSelectorI *` **`makeClone`**`(IloEnvI * env) const`

This member function is called internally to duplicate the current branch selector.

Class IloCsvReader::IloColumnHeaderNotFoundException

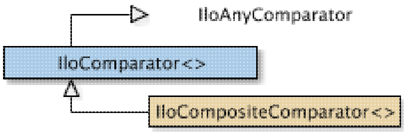
Definition file: ilconcert/ilocsvreader.h

Exception thrown for unfound header.
This exception is thrown by the member functions listed below if a header (column name) that you use does not exist.

- IloCsvLine::getFloatByHeader
- IloCsvLine::getIntByHeader
- IloCsvLine::getStringByHeader
- IloCsvLine::getFloatByHeaderOrDefaultValue
- IloCsvLine::getIntByHeaderOrDefaultValue
- IloCsvLine::getStringByHeaderOrDefaultValue
- IloCsvReader::getPosition
- IloCsvTableReader::getPosition

Class IloComparator<>

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>



A comparator is a class that implements a comparison between two objects. `IloComparator` is a template class, meaning that it can be instantiated to define a comparator for user-defined classes.

A comparator can be constructed from a single evaluator (see member functions `IloEvaluator::makeLessThanComparator`, `IloEvaluator::makeGreaterThanComparator`), from a composition of evaluators (`IloLexicographicComparator`, `IloParetoComparator`), or by using macros (`ILOCOMPARATOR0` and `ILOCLIKECOMPARATOR0`).

There are two main comparison functions of a comparator: `operator()` and `isBetterThan`.

The function `operator()` takes two objects, `o1` and `o2`, and by default returns -1, 0, or 1 depending on whether `o1` is respectively better, equal or worse than `o2`. The optional argument `nu` is a context that can be used for the comparison:

```
IloInt operator()(IloObject o1, IloObject o2, IloAny nu = 0) const;
```

The function `isBetterThan` takes two objects, `o1` and `o2`, and returns `IloTrue` if and only if `o1` is preferred to `o2`. The optional argument `nu` is a context that can be used for the comparison:

```
IloBool isBetterThan(IloObject o1, IloObject o2, IloAny nu = 0) const;
```

It is possible to combine comparators and evaluators into more complicated comparators. See `IloLexicographicComparator` and `IloParetoComparator`.

For more information, see Selectors.

See Also: IloEvaluator, IloLexicographicComparator, IloParetoComparator, ILOCOMPARATOR0, ILOCLIKECOMPARATOR0

Method Summary	
`public IloBool`	`isBetterOrEqual(IloObject left, IloObject right, IloAny nu=0) const` Returns `IloTrue` when the left-hand side is better than or equal to the right-hand side.
`public IloBool`	`isBetterThan(IloObject left, IloObject right, IloAny nu=0) const` Returns `IloTrue` when the left-hand side is better than the right-hand side.
`public IloBool`	`isEqual(IloObject left, IloObject right, IloAny nu=0) const` Returns `IloTrue` if and only if the left-hand side is of equal quality to the right-hand side.

public IloBool	isWorseOrEqual(IloObject left, IloObject right, IloAny nu=0) const Returns IloTrue when the left-hand side is worse than or equal to the right-hand side.
public IloBool	isWorseThan(IloObject left, IloObject right, IloAny nu=0) const Returns IloTrue when the left-hand side is worse than the right-hand side.
public IloComparator< IloObject >	makeInverse() const Inverts the sense of a comparator. The returned comparator compares as the invoking comparator, but with the objects being compared interchanged.
public IloInt	operator()(IloObject left, IloObject right, IloAny nu=0) const

Methods

public IloBool **isBetterOrEqual**(IloObject left, IloObject right, IloAny nu=0) const

Returns IloTrue when the left-hand side is better than or equal to the right-hand side.

This member function returns IloTrue if and only if the the left-hand side object left is better than or equal to the right-hand side object right. The parameter nu is a context that will be passed to the evaluator(s) of the comparator.

the comparator

public IloBool **isBetterThan**(IloObject left, IloObject right, IloAny nu=0) const

Returns IloTrue when the left-hand side is better than the right-hand side.
This member function returns IloTrue if and only if the the left-hand side object left is better than the right-hand side object right. The parameter nu is a context that will be passed to the evaluator(s) of the comparator.

the comparator

public IloBool **isEqual**(IloObject left, IloObject right, IloAny nu=0) const

Returns IloTrue if and only if the left-hand side is of equal quality to the right-hand side.
This member function returns IloTrue if and only if the the left-hand side object left is equal to the right-hand side object right. The parameter nu is a context that will be passed to the evaluator(s) of the comparator.

the comparator

public IloBool **isWorseOrEqual**(IloObject left, IloObject right, IloAny nu=0) const

Returns IloTrue when the left-hand side is worse than or equal to the right-hand side.
This member function returns IloTrue if and only if the the left-hand side object left is worse than or equal to the right-hand side object right. The parameter nu is a context that will be passed to the evaluator(s) of the comparator.

the comparator

public IloBool **isWorseThan**(IloObject left, IloObject right, IloAny nu=0) const

Returns IloTrue when the left-hand side is worse than the right-hand side.

This member function returns `IloTrue` if and only if the the left-hand side object `left` is worse than the right-hand side object `right`. The parameter `nu` is a context that will be passed to the evaluator(s) of the comparator.

the comparator

```
public IloComparator< IloObject > makeInverse() const
```

Inverts the sense of a comparator. The returned comparator compares as the invoking comparator, but with the objects being compared interchanged.
This member function returns a comparator that inverts the sense of the invoking comparator. The returned comparator interchanges the objects to be compared.

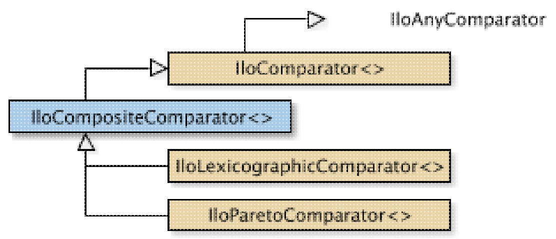
```
public IloInt operator()(IloObject left, IloObject right, IloAny nu=0) const
```

This member function implements the comparison of two objects based on the evaluator with which the invoking comparator was created and based on the sense of comparison of the invoking comparator. This member function returns -1, 0, or 1 depending on whether object `left` is considered to be better, equal, or worse than object `right`. The parameter `nu` is a context that will be passed to the evaluator(s) of the comparator.

the comparator

Class IloCompositeComparator<>

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>



A composite comparator is a comparator that relies on a list of subordinate comparators (for example, a lexicographical or a Pareto comparator). The member function `add` allows you to add a new subordinate comparator to this list.

For more information, see Selectors.

See Also: IloParetoComparator, IloLexicographicComparator

Method Summary	
public void	add(IloComparator< IloObject > comparator)

Inherited Methods from `IloComparator`
isBetterOrEqual, isBetterThan, isEqual, isWorseOrEqual, isWorseThan, makeInverse, operator()

Methods

public void **add**(IloComparator< IloObject > comparator)

This member function adds a new comparator `comparator` to the end of the list of comparators of the composite comparator.

Class IloCondition

Definition file: ilconcert/ilothread.h

Provides synchronization primitives adapted to Concert Technology for use in a parallel application.
The class `IloCondition` provides synchronization primitives adapted to Concert Technology for use in a parallel application.

An instance of the class `IloCondition` allows several threads to synchronize on a specific event. In this context, inter-thread communication takes place through signals. A thread expecting a condition of the computation state (say, `conditionC`) to be true before it executes a `treatmentT` can wait until the condition is true. When computation reaches a state where `conditionC` holds, then another thread can signal this fact by notifying a single waiting thread or by broadcasting to all the waiting threads that `conditionC` has now been met.

The conventional template for waiting on `conditionC` looks like this:

```
mutex.lock();
while (conditionC does not hold)
       condition.wait(&mutex);
doTreatmentT();
mutex.unlock();
```

That template has the following properties:

- The whole fragment is a critical section so that the evaluation of `conditionC` is protected. (Indeed, it would be unsafe to evaluate `conditionC` while at the same time another thread modifies the computation state and affects the truth value of `conditionC`.) The pair of member functions `IloFastMutex::lock` and `IloFastMutex::unlock` delimit the critical section.
- When a thread enters the `wait` call, the mutex is automatically unlocked by the system.
- The loop that repeatedly checks `conditionC` is essential to the correctness of the code fragment. It protects against the following possibility: between the time that a thread modifies the computation state (so that `conditionC` holds) and notifies a waiting thread and the moment the waiting thread wakes up, the computation state might have been changed by another thread, and `conditionC` might very well be false.
- Upon returning from the `wait` call, the mutex is locked. The operation of waking up and locking the mutex is atomic. In other words, nothing can happen between the waking and the locking.

System Class

`IloCondition` is a system class.

Most Concert Technology classes are actually handle classes whose instances point to objects of a corresponding implementation class. For example, instances of the Concert Technology class `IloNumVar` are handles pointing to instances of the implementation class `IloNumVarI`. Their allocation and de-allocation on the Concert Technology heap are managed by an instance of `IloEnv`.

However, system classes, such as `IloCondition`, differ from that Concert Technology pattern. `IloCondition` is an ordinary C++ class. Its instances are allocated on the C++ heap.

Instances of `IloCondition` are not automatically de-allocated by a call to `IloEnv::end`. You must explicitly destroy instances of `IloCondition` by means of a call to the delete operator (which calls the appropriate destructor) when your application no longer needs instances of this class.

Furthermore, you should not allocate—neither directly nor indirectly—any instance of `IloCondition` on the Concert Technology heap because the destructor for that instance of `IloCondition` will never be called automatically by `IloEnv::end` when it cleans up other Concert Technology objects on the Concert Technology heap.

For example, it is not a good idea to make an instance of `IloCondition` part of a conventional Concert Technology model allocated on the Concert Technology heap because that instance will not automatically be de-allocated from the Concert Technology heap along with the other Concert Technology objects.

De-allocating Instances of IloCondition

Instances of `IloCondition` differ from the usual Concert Technology objects because they are not allocated on the Concert Technology heap, and their de-allocation is not managed automatically for you by `IloEnv::end`. Instead, you must explicitly destroy instances of `IloCondition` by calling the delete operator when your application no longer needs those objects.

See Also: IloFastMutex

Constructor and Destructor Summary	
public	IloCondition()
public	~IloCondition()

Method Summary	
public void	broadcast()
public void	notify()
public void	wait(IloFastMutex * m)

Constructors and Destructors

`public` **`IloCondition`**`()`

This constructor creates an instance of `IloCondition` and allocates it on the C++ heap (not in a Concert Technology environment). The instance contains data structures specific to an operating system.

`public` **`~IloCondition`**`()`

The delete operator calls this destructor to de-allocate an instance of `IloCondition`. This destructor is called automatically by the runtime system. The destructor de-allocates data structures (specific to an operating system) of the invoking condition.

Methods

`public void` **`broadcast`**`()`

This member function wakes all threads currently waiting on the invoking condition. If there are no threads waiting, this member function does nothing.

`public void` **`notify`**`()`

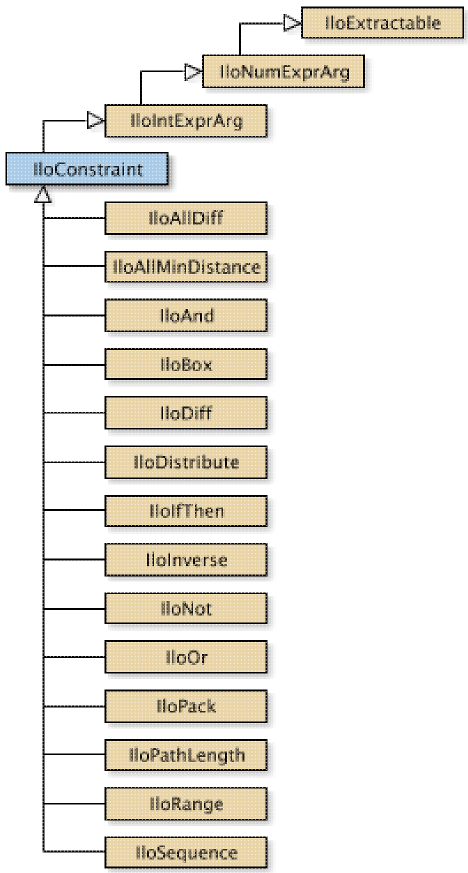
This member function wakes one of the threads currently waiting on the invoking condition.

`public void` **`wait`**`(IloFastMutex * m)`

This member function first puts the calling thread to sleep while it unlocks the mutex `m`. Then, when either of the member functions `broadcast` or `notify` wakes up that thread, this member function acquires the lock on `m` and returns.

Class IloConstraint

Definition file: ilconcert/iloexpression.h



An instance of this class is a constraint in a model.
To create a constraint, you can:

- use a constructor from a subclass of `IloConstraint`, such as `IloRange`, `IloAllDiff`, etc. For example:

```
IloAllDiff allDiff(env, vars);
```

- use a logical operator between constraints to return a constraint. For example, you can use the logical operators on other constraints, like this:

```
IloOr myOr = myConstraint1 || myConstraint2;
```

- use an arithmetic operator between a numeric variable and an expression to return a constraint. For example, you can use the arithmetic operators on numeric variables or expressions, like this:

```
IloRange rng = ( x + 3*y <= 7 );
```

After you create a constraint, you must explicitly add it to the model in order for it to be taken into account. To do so, use the member function `IloModel::add` or the template `IloAdd`. Then extract the model for an algorithm with the member function `IloAlgorithm::extract`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloConstraintArray, IloModel, IloRange

Constructor Summary	
public	IloConstraint()
public	IloConstraint(IloConstraintI * impl)

Method Summary	
public IloConstraintI *	getImpl() const

Inherited Methods from `IloIntExprArg`
getImpl

Inherited Methods from `IloNumExprArg`
getImpl

Inherited Methods from `IloExtractable`
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Constructors

public **IloConstraint**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IloConstraint**(IloConstraintI * impl)

This constructor creates a handle object from a pointer to an implementation object.

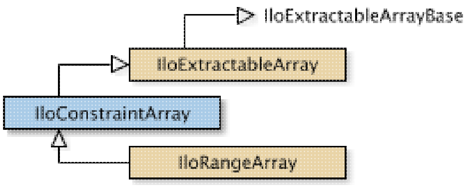
Methods

public IloConstraintI * **getImpl**() const

This member function returns a pointer to the implementation object of the invoking handle.

Class IloConstraintArray

Definition file: ilconcert/iloexpression.h



The array class of constraints for a model.
For each basic type, Concert Technology defines a corresponding array class. `IloConstraintArray` is the array class of constraints for a model.

Instances of `IloConstraintArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Arrays

See Also: IloConstraint, operator>>, operator<<

Constructor Summary	
`public`	`IloConstraintArray(IloDefaultArrayI * i=0)`
`public`	`IloConstraintArray(const IloConstraintArray & copy)`
`public`	`IloConstraintArray(const IloEnv env, IloInt n=0)`

Method Summary	
`public void`	`add(IloInt more, const IloConstraint x)`
`public void`	`add(const IloConstraint x)`
`public void`	`add(const IloConstraintArray x)`
`public IloConstraint`	`operator[](IloInt i) const`
`public IloConstraint &`	`operator[](IloInt i)`

Inherited Methods from `IloExtractableArray`
`add, add, add, endElements, setNames`

Constructors

`public` **`IloConstraintArray`**`(IloDefaultArrayI * i=0)`

This constructor creates an empty array. You cannot create instances of the undocumented class `IloDefaultArrayI`. As an argument in this default constructor, it allows you to pass 0 (zero) as a value to an optional argument in functions and member functions that accept an array as an argument.

`public` **`IloConstraintArray`**`(const IloConstraintArray & copy)`

This copy constructor makes a copy of the array specified by `copy`.

`public` **`IloConstraintArray`**`(const IloEnv env, IloInt n=0)`

This constructor creates an array of `n` elements, each of which is an empty handle.

Methods

`public void` **`add`**`(IloInt more, const IloConstraint x)`

This member function appends `constraint` to the invoking array multiple times. The argument `more` specifies how many times.

`public void` **`add`**`(const IloConstraint x)`

This member function appends `constraint` to the invoking array.

`public void` **`add`**`(const IloConstraintArray x)`

This member function appends the elements in `array` to the invoking array.

`public IloConstraint` **`operator[]`**`(IloInt i) const`

This operator returns a reference to the constraint located in the invoking array at the position specified by the index `i`. On `const` arrays, Concert Technology uses the `const` operator:

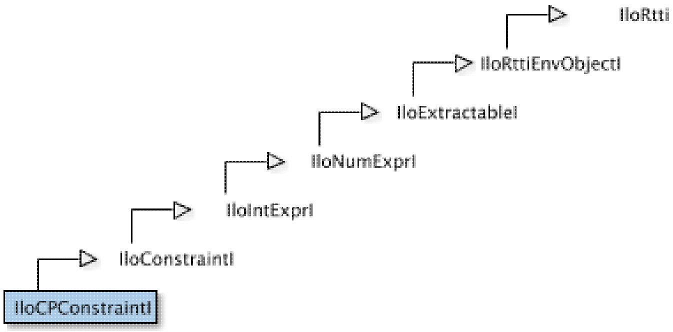
```
IloConstraint  operator[] (IloInt i) const;
```

`public IloConstraint &` **`operator[]`**`(IloInt i)`

This operator returns a reference to the constraint located in the invoking array at the position specified by the index `i`.

Class IloCPConstraintI

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>



The class `IloCPConstraintI` simplifies writing a new constraint using IBM® ILOG® Concert Technology. You can use the macro ILOCPCONSTRAINTWRAPPER to create a subclass of the class `IloCPConstraintI` with up to 4 data members.

If you want to create a constraint class with more than 4 data members, you must explicitly define a subclass of the class `IloCPConstraintI`. To do this, you must do the following:

- define the subclass
- add the macro `ILOCPCONSTRAINTWRAPPERDECL` in the class definition
- write the constructor
- add the macro `ILOCPCONSTRAINTWRAPPERIMPL` in the source file
- define the virtual member function `makeClone`
- define a function which creates an instance of the new subclass and returns a handle of this object (You must make a call to the static method `initTypeIndex` once before using the generated constraint. Therefore, it is recommended to include this call in the function.)
- define the virtual member function `extract`.

You can use the example in ILOCPCONSTRAINTWRAPPER as a model for writing your own new constraint explicitly.

See Also: ILOCPCONSTRAINTWRAPPER0

Constructor and Destructor Summary	
`public`	`IloCPConstraintI(IloEnvI *, const char *)`
`public`	`~IloCPConstraintI()`

Method Summary	
`public virtual void`	`display(ostream & out) const`
`public virtual IlcConstraint`	`extract(const IloSolver solver) const`
`public virtual IloExtractableI *`	`makeClone(IloEnvI * env) const`
`public void`	`use(const IloSolver solver, const IloExtractableArray extarray) const`
`public void`	`use(const IloSolver solver, const IloExtractable ext) const`

Constructors and Destructors

public **IloCPConstraintI**(IloEnvI *, const char *)

This constructor creates an instance of the class `IloCPConstraintI`. This constructor is called automatically in the constructor of its subclasses.

public **~IloCPConstraintI**()

This destructor is called automatically by the destructor of its subclasses. It frees memory used by the invoking object.

Methods

public virtual void **display**(ostream & out) const

This member function prints the invoking constraint on an output stream.

public virtual IlcConstraint **extract**(const IloSolver solver) const

In general terms, in Concert Technology, the objects of a model must be extracted for an algorithm (an instance of one of the subclasses of `IloAlgorithm`, such as `IloSolver`). This member function returns the internal constraint extracted for `solver` from the invoking constraint of a model.

public virtual IloExtractableI * **makeClone**(IloEnvI * env) const

This member function is called internally to duplicate the current constraint.

public void **use**(const IloSolver solver, const IloExtractableArray extarray) const

This member function forces the extraction of an array of extractables and its subextractables.

public void **use**(const IloSolver solver, const IloExtractable ext) const

This member function forces the extraction of an extractable and its subextractables.

Class IloCPTrace

Definition file: ilsolver/ilosolverhandle.h
Include file: <ilsolver/ilosolver.h>

An instance of this class is used to wrap the Solver trace mechanism. It enables you to set a trace at the start of a search.

See Also: IloCPTraceI, ILOCPTRACEWRAPPER0

Constructor Summary	
`public`	`IloCPTrace()`
`public`	`IloCPTrace(IloCPTraceI * impl=0)`

Method Summary	
`public IloCPTraceI *`	`getImpl() const`

Constructors

`public` **`IloCPTrace`**`()`

This constructor creates a trace at the start of search.

`public` **`IloCPTrace`**`(IloCPTraceI * impl=0)`

This constructor creates a trace from its implementation object.

Methods

`public IloCPTraceI *` **`getImpl`**`() const`

This member function returns a pointer to the implentation object of the invoking trace. This member function is useful when you need to be sure that you are using the same copy of the invoking trace in more than one situation.

Class IloCPTraceI

Definition file: ilsolver/ilosolverhandle.h
Include file: <ilsolver/ilosolver.h>

An instance of this class is used to wrap the Solver trace mechanism. It enables you to set a trace at the start of a search.

A trace wrapper is implemented by means of two classes: a handle class and an implementation class. In other words, an instance of the class `IloCPTrace` (a handle) contains a data member (the handle pointer) that points to an instance of the class `IloCPTraceI` (its implementation object).

See Also: IloCPTrace, ILOCPTRACEWRAPPER0

Constructor and Destructor Summary	
`public`	`IloCPTraceI()`
`public`	`~IloCPTraceI()`

Method Summary	
`public virtual void`	`execute(const IloSolver solver) const`

Constructors and Destructors

`public` **`IloCPTraceI`**`()`

This constructor creates a trace at the start of a search.

`public` **`~IloCPTraceI`**`()`

This destructor is called automatically by the destructor of its subclasses. It frees memory used by the objects.

Methods

`public virtual void` **`execute`**`(const IloSolver solver) const`

Solver calls this member function when it starts the search.

If you want to use this facility, you must define this virtual member function. This member function should not be called at the same time as any other side effects of the search, such as filter levels.

Class IloCsvLine

Definition file: ilconcert/ilocsvreader.h

Represents a line in a csv file.
An instance of `IloCsvLine` represents a single line in a file of comma-separated values (csv file).

Constructor and Destructor Summary	
public	`IloCsvLine()`
public	`IloCsvLine(IloCsvLineI * impl)`
public	`IloCsvLine(const IloCsvLine & csvLine)`

Method Summary	
public void	`copy(const IloCsvLine)`
public IloBool	`emptyFieldByHeader(const char * name) const`
public IloBool	`emptyFieldByPosition(IloInt i) const`
public void	`end()`
public IloNum	`getFloatByHeader(const char * name) const`
public IloNum	`getFloatByHeaderOrDefaultValue(const char * name, IloNum defaultValue) const`
public IloNum	`getFloatByPosition(IloInt i) const`
public IloNum	`getFloatByPositionOrDefaultValue(IloInt i, IloNum defaultValue) const`
public IloCsvLineI *	`getImpl() const`
public IloInt	`getIntByHeader(const char * name) const`
public IloInt	`getIntByHeaderOrDefaultValue(const char * name, IloInt defaultValue) const`
public IloInt	`getIntByPosition(IloInt i) const`
public IloInt	`getIntByPositionOrDefaultValue(IloInt i, IloInt defaultValue) const`
public IloInt	`getLineNumber() const`
public IloInt	`getNumberOfFields() const`
public char *	`getStringByHeader(const char * name) const`
public char *	`getStringByHeaderOrDefaultValue(const char * name, const char * defaultValue) const`
public char *	`getStringByPosition(IloInt i) const`
public char *	`getStringByPositionOrDefaultValue(IloInt i, const char * defaultValue) const`
public void	`operator=(const IloCsvLine & csvLine)`
public IloBool	`printValueOfKeys() const`

Constructors and Destructors

`public IloCsvLine()`

This constructor creates a csv line object whose handle pointer is null. This object must be assigned before it can be used.

`public IloCsvLine(IloCsvLineI * impl)`

This constructor creates a handle object (an instance of `IloCsvLine`) from a pointer to an implementation object (an instance of the class `IloCsvLineI`).

`public IloCsvLine(const IloCsvLine & csvLine)`

This copy constructor creates a handle from a reference to a csv line object. The csv line object and `csvLine` both point to the same implementation object.

Methods

`public void copy(const IloCsvLine)`

This member function returns the real number of the invoking csv line in the data file.

`public IloBool emptyFieldByHeader(const char * name) const`

This member function returns `IloTrue` if the field denoted by the string `name` in the invoking csv line is empty. Otherwise, it returns `IloFalse`

`public IloBool emptyFieldByPosition(IloInt i) const`

This member function returns `IloTrue` if the field denoted by `i` in the invoking csv line is empty. Otherwise, it returns `IloFalse`

`public void end()`

This member function deallocates the memory used by the csv line. If you no longer need a csv line, you can call this member function to reduce memory consumption.

`public IloNum getFloatByHeader(const char * name) const`

This member function returns the float contained in the field `name` in the invoking csv line.

If you have a loop in which you are getting a string, integer, or float by header on several lines with the same header name, it is better for performance to get the position of the header named `name` using the member function `IloCsvReader::getPosition(name)` than using `IloCsvLine::getFloatByPosition` (position of name in the header line).

`public IloNum getFloatByHeaderOrDefaultValue(const char * name, IloNum defaultValue) const`

This member function returns the float contained in the field `name` in the invoking csv line if this field contains a value. Otherwise, it returns `defaultValue`.

`public IloNum getFloatByPosition(IloInt i) const`

This member function returns the float contained in the field `i` in the invoking csv line.

`public IloNum getFloatByPositionOrDefaultValue(IloInt i, IloNum defaultValue) const`

This member function returns the float contained in the field `i` in the invoking csv line if this field contains a value. Otherwise, it returns `defaultValue`.

`public IloCsvLineI * getImpl() const`

This member function returns a pointer to the implementation object corresponding to the invoking csv line.

`public IloInt getIntByHeader(const char * name) const`

This member function returns the integer contained in the field `name` in the invoking csv line.

If you have a loop in which you are getting a string, integer, or float by header on several lines with the same header name, it is better for performance to get the position of the header named `name` using the member function `IloCsvReader::getPosition(name)` than using `IloCsvLine::getIntByPosition` (position of name in the header line).

`public IloInt getIntByHeaderOrDefaultValue(const char * name, IloInt defaultValue) const`

This member function returns the integer contained in the field `name` in the invoking csv line if this field contains a value. Otherwise, it returns `defaultValue`.

`public IloInt getIntByPosition(IloInt i) const`

This member function returns the integer contained in the field `i` in the invoking csv line.

`public IloInt getIntByPositionOrDefaultValue(IloInt i, IloInt defaultValue) const`

This member function returns the integer contained in the field `i` in the invoking csv line if this field contains a value. Otherwise, it returns `defaultValue`.

`public IloInt getLineNumber() const`

This member function returns the real number of the invoking csv line in the data file.

`public IloInt` **`getNumberOfFields`**`() const`

This member function returns the number of fields in the line.

`public char *` **`getStringByHeader`**`(const char * name) const`

This member function returns a reference to the string contained in the field `name` in the invoking csv line.

If you have a loop in which you are getting a string, integer, or float by header on several lines with the same header name, it is better for performance to get the position of the header named `name` using the member function `IloCsvReader::getPosition(name)` than using `IloCsvLine::getStringByPosition` (position of name in the header line).

`public char *` **`getStringByHeaderOrDefaultValue`**`(const char * name, const char * defaultValue) const`

This member function returns the string contained in the field `name` in the invoking csv line if this field contains a value. Otherwise, it returns `defaultValue`.

`public char *` **`getStringByPosition`**`(IloInt i) const`

This member function returns a reference to the string contained in the field number `i` in the invoking csv line.

`public char *` **`getStringByPositionOrDefaultValue`**`(IloInt i, const char * defaultValue) const`

This member function returns the string contained in the field `i` in the invoking csv line if this field contains a value. Otherwise, it returns `defaultValue`.

`public void` **`operator=`**`(const IloCsvLine & csvLine)`

This operator assigns an address to the handle pointer of the invoking csv line. This address is the location of the implementation object of the argument `csvLine`.

After execution of this operator, the invoking csv line and `csvLine` both point to the same implementation object.

`public IloBool` **`printValueOfKeys`**`() const`

This member function prints the values of the keys fields in this line.

Class IloCsvReader

Definition file: ilconcert/ilocsvreader.h

Reads a formatted csv file.
An instance of `IloCsvReader` reads a file of comma-separated values of a specified format. The csv file can be a multitable or a single table file. Empty lines and commented lines are allowed everywhere in the file.

Format of multitable files

The first column of the table must contain the name of the table.

Each table can begin with a line containing column headers, the first field of this line must have this format: `tableName|NAMES`

The keys can be specified in the data file by adding a line at the beginning of the table. This line is formatted as follows:

- the first field is `tableName|KEYS`
- the other fields have the value 1 if the corresponding column is a key for the table; if not they have the value 0.

If this line doesn't exist, all columns form a key. If you need to get a line having a specific value for a field, you must add the key line in which you specify that this field is a key for the table.

Any line containing '|' in its first field is ignored by the reader.

A table can be split in several parts in the file (for example, you have a part of table TA, then table TB, then the end of table TA).

Example

```
NODES|NAMES,node_type,node_name,xcoord,ycoord
NODES|KEYS,1,1,0,0
NODES,1,node1,0,1
NODES,1,node2,0,2
NODES,2,node1,0,4
```

Format of single table files

The line containing the column headers, if it exists, must have a first field of the following format: `Field|NAMES`.

Table keys can be specified by adding a line at the beginning of the table. This line must have a first field with this format: `tableName|KEYS`. If this line doesn't exist, all columns form a key.

Example

```
Field|NAMES,nodeName,xCoord,yCoord
Field|KEYS,1,0,0
node1,0,1
node2,0,2
```

Constructor and Destructor Summary	
public	IloCsvReader()
public	IloCsvReader(IloCsvReaderI * impl)

public	IloCsvReader(const IloCsvReader & csv)
public	IloCsvReader(IloEnv env, const char * problem, IloBool multiTable=IloFalse, IloBool allowTableSplitting=IloFalse, const char * separator=",;\t", const char decimalp='.', const char quote='\"', const char comment='#')

Method Summary	
public void	end()
public IloNum	getCsvFormat()
public IloCsvLine	getCurrentLine() const
public IloEnv	getEnv() const
public IloNum	getFileVersion()
public IloCsvReaderI *	getImpl() const
public IloCsvLine	getLineByKey(IloInt numberOfKeys, const char *, ...)
public IloCsvLine	getLineByNumber(IloInt i)
public IloInt	getNumberOfColumns()
public IloInt	getNumberOfItems()
public IloInt	getNumberOfKeys() const
public IloInt	getNumberOfTables()
public IloInt	getPosition(const char * headingName) const
public IloCsvTableReader	getReaderForUniqueTableFile() const
public const char *	getRequiredBy()
public IloCsvTableReader	getTable()
public IloCsvTableReader	getTableByName(const char * name)
public IloCsvTableReader	getTableByNumber(IloInt i)
public IloBool	isHeadingExists(const char * headingName) const
public void	operator=(const IloCsvReader & csv)
public IloBool	printKeys() const

Inner Class	
IloCsvReader::IloColumnHeaderNotFoundException	Exception thrown for unfound header.
IloCsvReader::IloCsvReaderParameterException	Exception thrown for incorrect arguments in constructor.
IloCsvReader::IloDuplicatedTableException	Exception thrown for tables of same name in csv file.
IloCsvReader::IloFieldNotFoundException	Exception thrown for field not found.
IloCsvReader::IloFileNotFoundException	Exception thrown when file is not found.
IloCsvReader::IloIncorrectCsvReaderUseException	Exception thrown for call to inappropriate csv reader.
IloCsvReader::IloLineNotFoundException	Exception thrown for unfound line.
IloCsvReader::IloTableNotFoundException	Exception thrown for unfound table.
IloCsvReader::LineIterator	Line-iterator for csv readers.
IloCsvReader::TableIterator	Table-iterator of csv readers.

Constructors and Destructors

public **IloCsvReader**()

This constructor creates a csv reader object whose handle pointer is null. This object must be assigned before it can be used.

```
public IloCsvReader(IloCsvReaderI * impl)
```

This constructor creates a handle object (an instance of `IloCsvReader`) from a pointer to an implementation object (an instance of the class `IloCsvReaderI`).

```
public IloCsvReader(const IloCsvReader & csv)
```

This copy constructor creates a handle from a reference to a csv reader object. Both the csv reader object and `csv` point to the same implementation object.

```
public IloCsvReader(IloEnv env, const char * problem, IloBool multiTable=IloFalse,
IloBool allowTableSplitting=IloFalse, const char * separator=",;\t", const char
decimalp='.', const char quote='\"', const char comment='#')
```

This constructor creates a csv reader object for the file `problem` in the environment `env`. If the argument `isCached` has the value `IloTrue`, the data of the file will be stored in the memory.

The cached mode is useful only if you need to read lines by keys. It needs consequent memory consumption and takes time to load data according to the csv file size.

If the argument `isMultiTable` has the value `IloTrue`, the file `problem` is read as a multitable file. The default value is `IloFalse`.

If the argument `allowTableSplitting` has the value `IloFalse`, splitting the table into several parts in the file is not permitted. The default value is `IloFalse`.

The string `separator` represents the characters used as separator in the data file. The default values are `, ; t`.

The character `decimal` represents the character used to write decimal numbers in the data file. The default value is `.` (period).

The character `quote` represents the character used to quote expressions.

The character `comment` represents the character used at the beginning of each commented line. The default value is `#`.

Methods

```
public void end()
```

This member function deallocates the memory used by the csv reader. If you no longer need a csv reader, you can reduce memory consumption by calling this member function.

```
public IloNum getCsvFormat()
```

This member function returns the format of the csv data file. This format is identified in the data file by `ILOG_CSV_FORMAT`.

Example

```
ILOG_CSV_FORMAT;1
```

`getCsvFormat()` returns `1`.

> **Note**
>
> This member function can be used only if `isMultiTable` has the value `IloTrue`.

`public IloCsvLine` **`getCurrentLine`**`() const`

This member function returns the last line read by `getLineByKey` or `getLineByNumber`.

> **Note**
>
> This member function can be used only if `isMultiTable` has the value `IloFalse`.

`public IloEnv` **`getEnv`**`() const`

This member function returns the environment object corresponding to the invoking csv reader.

`public IloNum` **`getFileVersion`**`()`

This member function returns the version of the csv data file. This information is identified in the data file by `ILOG_DATA_SCHEMA`.

Example

```
ILOG_DATA_SCHEMA;PROJECTNAME;0.9
```

`getFileVersion()` returns `0.9`.

> **Note**
>
> This member function can be used only if `isMultiTable` has the value `IloTrue`.

`public IloCsvReaderI *` **`getImpl`**`() const`

This member function returns a pointer to the implementation object corresponding to the invoking csv reader.

`public IloCsvLine` **`getLineByKey`**`(IloInt numberOfKeys, const char *, ...)`

This member function takes `numberOfKeys` arguments; these arguments are used as one key to identify a line. It returns an instance of `IloCsvLine` representing the line having `(key1, key2, ...)` in the data file. If the number of keys specified is less than the number of keys in the table, this member function throws an exception. Each time `getLineByNumber` or `getLineByKey` is called, the previous line read by one of these methods is deleted.

Note

This member function can be used only if `isMultiTable` has the value `IloFalse`.

```
public IloCsvLine getLineByNumber(IloInt i)
```

This member function returns an instance of `IloCsvLine` representing the line numbered `i` in the data file. If `i` does not exist, this member function throws an exception. Each time `getLineByNumber` or `getLineByKey` is called, the previous line read by one of these methods is deleted.

Note

This member function can be used only if `isMultiTable` has the value `IloFalse`.

```
public IloInt getNumberOfColumns()
```

This member function returns the number of columns in the table. If the first column contains the name of the table it is ignored.

Note

This member function can be used only if `isMultiTable` has the value `IloFalse`.

```
public IloInt getNumberOfItems()
```

This member function returns the number of lines of the table excluding blank lines, commented lines, and the header line.

Note

This member function can be used only if `isMultiTable` has the value `IloFalse`.

```
public IloInt getNumberOfKeys() const
```

This member function returns the number of keys for the table.

Note

This member function can be used only if `isMultiTable` has the value `IloFalse`.

```
public IloInt getNumberOfTables()
```

This member function returns the number of tables in the data file.

```
public IloInt getPosition(const char * headingName) const
```

This member function returns the position (column number) of the `headingName` in the file.

> **Note**
>
> This member function can be used only if `isMultiTable` has the value `IloFalse`.

`public IloCsvTableReader` **`getReaderForUniqueTableFile`**`() const`

This member function returns an `IloCsvTableReader` for the unique table contained in the csv data file.

> **Note**
>
> This member function can be used only if `isMultiTable` has the value `IloFalse`.

`public const char *` **`getRequiredBy`**`()`

This member function returns the name of the project that uses the csv data file. This information is identified in the data file by `ILOG_DATA_SCHEMA`.

Example

```
ILOG_DATA_SCHEMA;PROJECTNAME;0.9
```

`getRequiredBy()` returns `PROJECTNAME`.

> **Note**
>
> This member function can be used only if `isMultiTable` has the value `IloTrue`.

`public IloCsvTableReader` **`getTable`**`()`

This member function returns an instance of `IloCsvTableReader` representing the unique table in the data file.

> **Note**
>
> This member function can be used only if `isMultiTable` has the value `IloFalse`.

`public IloCsvTableReader` **`getTableByName`**`(const char * name)`

This member function returns an instance of `IloCsvTableReader` representing the table named `name` in the data file.

> **Note**
>
> This member function can be used only if `isMultiTable` has the value `IloTrue`.

`public IloCsvTableReader` **`getTableByNumber`**`(IloInt i)`

This member function returns an instance of `IloCsvTableReader` representing the table numbered `i` in the data file.

Note
This member function can be used only if `isMultiTable` has the value `IloTrue`.

```
public IloBool isHeadingExists(const char * headingName) const
```

This member function returns `IloTrue` if the column header `headingName` exists. Otherwise, it returns `IloFalse`.

Note
This member function can be used only if `isMultiTable` has the value `IloFalse`.

```
public void operator=(const IloCsvReader & csv)
```

This operator assigns an address to the handle pointer of the invoking csv reader. This address is the location of the implementation object of the argument `csv`.

After execution of this operator, both the invoking csv reader and `csv` point to the same implementation object.

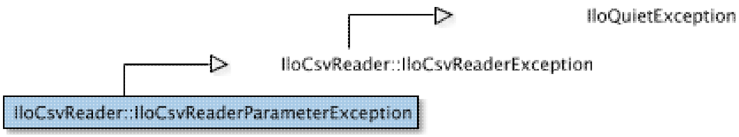
```
public IloBool printKeys() const
```

This member function prints the column header of keys if the header exists. Otherwise, it prints the column numbers of keys.

Note
This member function can be used only if `isMultiTable` has the value `IloFalse`.

Class IloCsvReader::IloCsvReaderParameterException

Definition file: ilconcert/ilocsvreader.h



Exception thrown for incorrect arguments in constructor.
This exception is thrown in the constructor of the csv reader if the argument values used in the csv reader constructor are incorrect.

Class IloCsvTableReader

Definition file: ilconcert/ilocsvreader.h

Reads a csv table with format.
An instance of `IloCsvTableReader` is used to read a table of comma-separated values (csv) with a specified format.

An instance is built using a pointer to an implementation class of `IloCsvReader`, which must be created first.

Constructor and Destructor Summary	
public	IloCsvTableReader()
public	IloCsvTableReader(IloCsvTableReaderI * impl)
public	IloCsvTableReader(const IloCsvTableReader & csv)
public	IloCsvTableReader(IloCsvReaderI * csvReaderImpl, const char * name=0)

Method Summary	
public void	end()
public IloCsvLine	getCurrentLine() const
public IloEnv	getEnv() const
public IloCsvTableReaderI *	getImpl() const
public IloCsvLine	getLineByKey(IloInt numberOfKeys, const char *, ...)
public IloCsvLine	getLineByNumber(IloInt i)
public const char *	getNameOfTable() const
public IloInt	getNumberOfColumns()
public IloInt	getNumberOfItems()
public IloInt	getNumberOfKeys() const
public IloInt	getPosition(const char *) const
public IloBool	isHeadingExists(const char * headingName) const
public void	operator=(const IloCsvTableReader & csv)
public IloBool	printKeys() const

Inner Class	
IloCsvTableReader::LineIterator	Line-iterator for csv table readers.

Constructors and Destructors

public **IloCsvTableReader**()

This constructor creates a table csv reader object whose handle pointer is null. This object must be assigned before it can be used.

public **IloCsvTableReader**(IloCsvTableReaderI * impl)

This constructor creates a handle object (an instance of `IloCsvReader`) from a pointer to an implementation object (an instance of the class `IloCsvReaderI`).

`public` **`IloCsvTableReader`**`(const IloCsvTableReader & csv)`

This copy constructor creates a handle from a reference to a table csv reader object.

The table csv reader object and `csv` both point to the same implementation object.

`public` **`IloCsvTableReader`**`(IloCsvReaderI * csvReaderImpl, const char * name=0)`

This constructor creates a table csv reader object using the implementation class of a csv reader `csvimpl`. The second argument is the name of the table.

Methods

`public void` **`end`**`()`

This member function deallocates the memory used by the table csv reader.

If you no longer need the table csv reader, calling this member function can reduce memory consumption.

`public IloCsvLine` **`getCurrentLine`**`() const`

This member function returns the last line read using `getLineByKey` or `getLineByNumber`.

`public IloEnv` **`getEnv`**`() const`

This member function returns the environment object corresponding to the invoking table csv reader.

`public IloCsvTableReaderI *` **`getImpl`**`() const`

This member function returns a pointer to the implementation object corresponding to the invoking table csv reader.

`public IloCsvLine` **`getLineByKey`**`(IloInt numberOfKeys, const char *, ...)`

This member function takes `numberOfKeys` arguments. These arguments are used as one key to identify a line. If the specified number of keys is less than the number of keys of the table, this member function throws an exception.

Otherwise, it returns an instance of `IloCsvLine` representing the line having `(key1, key2, ...)` in the data file.

`public IloCsvLine` **`getLineByNumber`**`(IloInt i)`

This member function returns an instance of `IloCsvLine` representing the line number `i` in the data file if it exists. Otherwise, it throws an exception.

Each time `getLineByNumber` or `getLineByKey` is called, the previous line read by one of those methods is deleted.

`public const char *` **`getNameOfTable`**`() const`

This member function returns the name of the table.

`public IloInt` **`getNumberOfColumns`**`()`

This member function returns the number of columns in the table. If the first column contains the name of the table, it is ignored.

`public IloInt` **`getNumberOfItems`**`()`

This member function returns the number of lines of the table excluding blank lines, commented lines, and the header line.

Note

This member function can be used only if `isMultiTable` has the value `IloFalse`.

`public IloInt` **`getNumberOfKeys`**`() const`

This member function returns the number of keys in the table.

`public IloInt` **`getPosition`**`(const char *) const`

This member function returns the position (column number) of `headingName` in the table.

`public IloBool` **`isHeadingExists`**`(const char * headingName) const`

This member function returns `IloTrue` if the column header named `headingName` exists. Otherwise, it returns `IloFalse`.

`public void` **`operator=`**`(const IloCsvTableReader & csv)`

This operator assigns an address to the handle pointer of the invoking table csv reader.

This address is the location of the implementation object of the argument `csv`.

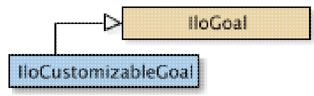
After execution of this operator, the invoking table csv reader and `csv` both point to the same implementation object.

```
public IloBool printKeys() const
```

This member function prints the column headers of keys if they exist. Otherwise, it prints the column numbers of keys.

Class IloCustomizableGoal

Definition file: ilsolver/custgoal.h
Include file: <ilsolver/ilosolver.h>



This class is a goal for instantiating an array of integer variables. It implements a family of strategies based on filters. It selects an uninstantiated variable `x` and a value `v` in the domain of `x` and creates the choice point `IlcOr(IlcSetValue(x, v), IlcRemoveValue(x, v))`. If the value returned by `IloCustomizableGoal::getBestGenerate` is `IloTrue`, it then selects another variable, otherwise it selects another value for the variable `x`. The goal succeeds when all variables of the array are instantiated.

To select variables and values, the goal uses an ordered list of filters for variables and an ordered list of filters for values.

In its simplest form, a filter for variables is defined by an evaluator taking an `IlcIntVar` as argument. A filter keeps the variables with the smallest evaluation and discards the others. The filters are applied in the order of addition to the `IloCustomizableGoal` instance. The first filter keeps the best variables. If several variables remain, the next filter is applied. If no more filters remain, the variable with the smallest index in the array given in the constructor is selected.

A value for the selected variable is selected by values filters. In its simplest form, a filter for values is defined by an evaluator taking an `IlcInt` (the value) as argument and an `IloIntVar` as a context (the selected variable is passed as a context). Filters for values operate in the same way as filters for variables. When several values remain after having applied all the filters, the smallest value is selected.

By default an instance of `IloCustomizableGoal` contains two filters for variables: the first one selects the variables with the largest impact and the second one selects randomly. It contains two selectors for values: the first one selects the value with the smallest impact and the second one selects randomly. When adding a filter for variables (or, respectively, values), the default filters for variables (or, respectively, values) are discarded.

More complex filters can be added to keep more variables than just the ones with the smallest evaluation. For an example of how to use `IloCustomizableGoal`, see *"Using Impacts during Search"* in the *IBM ILOG Solver User's Manual*.

See Also: IlcImpactValueEvaluator, IlcSuccessRateValueEvaluator, IlcRandomValueEvaluator, IlcImpactVarEvaluator, IlcSuccessRateVarEvaluator, IlcLocalImpactVarEvaluator, IlcRandomVarEvaluator, IlcSizeVarEvaluator, IlcDegreeVarEvaluator, IlcSizeOverDegreeVarEvaluator, IlcReductionVarEvaluator, IlcBranchImpactVarEvaluator

Constructor Summary	
public	IloCustomizableGoal(IloEnv env, IloIntVarArray x)

Method Summary	
public void	addAbsoluteToleranceFilter(IloEvaluator< IlcInt > e, IlcFloat tol)
public void	addAbsoluteToleranceFilter(IloEvaluator< IlcIntVar > e, IlcFloat tol)
public void	addFilter(IloEvaluator< IlcInt > e)
public void	addFilter(IloEvaluator< IlcIntVar > e)
public void	addMinNumberFilter(IloEvaluator< IlcInt > e, IlcFloat number)
public void	addMinNumberFilter(IloEvaluator< IlcIntVar > e, IlcFloat number)

public void	addMinProportionFilter(IloEvaluator< IlcInt > e, IlcFloat proportion)
public void	addMinProportionFilter(IloEvaluator< IlcIntVar > e, IlcFloat proportion)
public void	addRelativeToleranceFilter(IloEvaluator< IlcInt > e, IlcFloat tol)
public void	addRelativeToleranceFilter(IloEvaluator< IlcIntVar > e, IlcFloat tol)
public IlcBool	getBestGenerate()
public void	setBestGenerate(IloBool best)

Inherited Methods from `IloGoal`
end, getEnv, getImpl, getName, getObject, setName, setObject

Constructors

public **IloCustomizableGoal**(IloEnv env, IloIntVarArray x)

This constructor creates a customizable goal on the environment `env`. The parameter `x` is the array of variables to perform search on.

Methods

public void **addAbsoluteToleranceFilter**(IloEvaluator< IlcInt > e, IlcFloat tol)

This member function adds a filter for values that keeps the values that are at a distance of at most `tol` from the best evaluation.

public void **addAbsoluteToleranceFilter**(IloEvaluator< IlcIntVar > e, IlcFloat tol)

This member function adds a filter for variables that keeps the variables that are at a distance of at most `tol` from the best evaluation.

public void **addFilter**(IloEvaluator< IlcInt > e)

This member function adds a basic filter for values that keeps all the values with the smallest evaluation.

public void **addFilter**(IloEvaluator< IlcIntVar > e)

This member function adds a basic filter for variables that keeps all the variables with the smallest evaluation.

public void **addMinNumberFilter**(IloEvaluator< IlcInt > e, IlcFloat number)

This member function adds a filter for values that keeps at least `number` of the values with the smallest evaluation.

`public void` **addMinNumberFilter**`(IloEvaluator< IlcIntVar > e, IlcFloat number)`

This member function adds a filter for variables that keeps at least `number` of the variables with the smallest evaluation.

`public void` **addMinProportionFilter**`(IloEvaluator< IlcInt > e, IlcFloat proportion)`

This member function adds a filter for values that keeps at least a proportion `p` of the values with the smallest evaluation. The value `p` must be greater than 0 and less than or equal to 1.

`public void` **addMinProportionFilter**`(IloEvaluator< IlcIntVar > e, IlcFloat proportion)`

This member function adds a filter for variables that keeps at least a proportion `p` of the variables with the smallest evaluation. The value `p` must be greater than 0 and less than or equal to 1.

`public void` **addRelativeToleranceFilter**`(IloEvaluator< IlcInt > e, IlcFloat tol)`

This member function adds a filter for values that keeps the values that are at a relative distance of at most `tol` from the best evaluation.

`public void` **addRelativeToleranceFilter**`(IloEvaluator< IlcIntVar > e, IlcFloat tol)`

This member function adds a filter for variables that keeps the variables that are at a relative distance of at most `tol` from the best evaluation.

`public IlcBool` **getBestGenerate**`()`

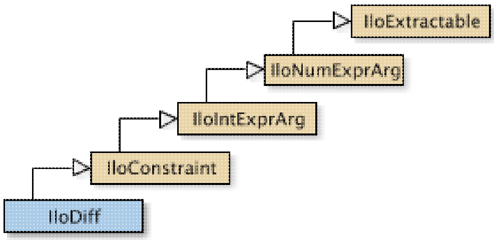
This member function returns the Boolean value set by `IloCustomizableGoal::setBestGenerate`.

`public void` **setBestGenerate**`(IloBool best)`

This member function turns the best generate strategy on and off for the invoking solver. If the value of best is `IloTrue`, the goal performs a best generate strategy: when an instantiation `x == a` fails, the goal again evaluates all variables to select a variable. If the value of best is `IloFalse`, the goal does not perform a best generate strategy: when an instantiation `x == a` fails, the goal tries another value for the same variable `x`. The default value is `IloFalse`.

Class IloDiff

Definition file: ilconcert/ilomodel.h



Constraint that enforces inequality.
An instance of this class is a constraint that enforces inequality (that is, “not equal” as specified by `!=`) in Concert Technology.

To create a constraint, you can:

- use the inequality `operator!=` on constrained variables (instances of `IloNumVar` and its subclasses) or expressions (instances of `IloExpr` and its subclasses).
- use a constructor from this class.

In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloAllDiff, IloConstraint, IloExpr, IloNumVar

Constructor Summary	
`public`	`IloDiff()`
`public`	`IloDiff(IloDiffI * impl)`
`public`	`IloDiff(const IloEnv env, const IloNumExprArg expr1, const IloNumExprArg expr2, const char * name=0)`
`public`	`IloDiff(const IloEnv env, const IloNumExprArg expr1, IloNum val, const char * name=0)`

Method Summary	
`public IloDiffI *`	`getImpl() const`

Inherited Methods from `IloConstraint`
`getImpl`

Inherited Methods from `IloIntExprArg`
`getImpl`

Inherited Methods from `IloNumExprArg`
`getImpl`

Inherited Methods from `IloExtractable`
`asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject`

Constructors

`public` **`IloDiff`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IloDiff`**`(IloDiffI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public` **`IloDiff`**`(const IloEnv env, const IloNumExprArg expr1, const IloNumExprArg expr2, const char * name=0)`

This constructor creates a constraint that enforces inequality (!=) in a model between the two expressions that are passed as its arguments. You must use the template `IloAdd` or the member function `IloModel::add` to add this constraint to a model in order for it to be taken into account.

The optional argument `name` is set to `0` by default.

`public` **`IloDiff`**`(const IloEnv env, const IloNumExprArg expr1, IloNum val, const char * name=0)`

This constructor creates a constraint that enforces inequality (!=) in a model between the expression `expr1` and the floating-point value that are passed as its arguments. You must use the template `IloAdd` or the member function `IloModel::add` to add this constraint to a model in order for it to be taken into account.

The optional argument `name` is set to `0` by default.

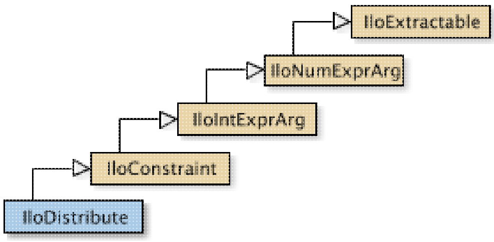
Methods

`public IloDiffI *` **`getImpl`**`() const`

This member function returns a pointer to the implementation object of the invoking handle.

Class IloDistribute

Definition file: ilconcert/ilomodel.h



For constraint programming:: a counting constraint in a model.
An instance of this class is a counting constraint in a model. You can use an instance of this class to count the number of occurrences of several values among the constrained variables in an array of constrained variables. You can also use an instance of this class to force the constrained variables of an array to assume values in such a way that only a limited number of the constrained variables assume each value.

For example, if we have five cars to paint in three available colors, then we might refer to the cars as `c1`, `c2`, `c3`, `c4`, `c5`, and the colors as `p1`, `p2`, `p3`. If we can allow no more than three cars to be painted `p1`, exactly three cars to be painted `p2`, and no more than one car to be painted `p3`, then we can represent our problem informally in terms of this constraint like this:

```
cards = [[0,3], [3,3], [0,1]]

values = [p1, p2, p3]

vars = [c1, c2, c3, c4, c5]
```

In more formal terms, the constrained variables in the array `cards` are equal to the number of occurrences in the array `vars` of the values in the array `values`. More precisely, for each `i`, `cards[i]` is equal to the number of occurrences of `values[i]` in the array `vars`. After propagation of this constraint, the minimum of `cards[i]` is at least equal to the number of variables contained in `vars` bound to the value at `values[i]`; and the maximum of `cards[i]` is at most equal to the number of variables contained in `vars` that contain the value at `values[i]` in their domain.

The arrays `cards` and `values` must be the same length; otherwise, Concert Technology throws an exception on platforms that support C++ exceptions when exceptions are enabled.

When an instance of this class is created by a constructor with only `cards` and `vars` as arguments (that is, there is no `values` argument), then the array of values that are being counted must be an array of consecutive integers starting with 0 (zero). In that case, for each `i`, `cards[i]` is equal to the number of occurrences of `i` in the array `vars`. After propagation of this constraint, the minimum of `cards[i]` is at least equal to the number of variables contained in `vars` bound to the value `i`; and the maximum of `cards[i]` is at most equal to the number of variables contained in `vars` that contain `i` in their domain.

In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloConstraint, IloSequence

Constructor Summary	
`public`	`IloDistribute()`

public	IloDistribute(IloDistributeI * impl)
public	IloDistribute(const IloEnv env, const IloIntVarArray cards, const IloIntArray values, const IloIntVarArray vars, const char * name=0)
public	IloDistribute(const IloEnv env, const IloIntVarArray cards, const IloIntVarArray vars, const char * name=0)

Method Summary	
public IloDistributeI *	getImpl() const

Inherited Methods from `IloConstraint`
getImpl

Inherited Methods from `IloIntExprArg`
getImpl

Inherited Methods from `IloNumExprArg`
getImpl

Inherited Methods from `IloExtractable`
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Constructors

public **IloDistribute**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IloDistribute**(IloDistributeI * impl)

This constructor creates a handle object from a pointer to an implementation object.

public **IloDistribute**(const IloEnv env, const IloIntVarArray cards, const IloIntArray values, const IloIntVarArray vars, const char * name=0)

This constructor creates a counting constraint in a model. You must use the template `IloAdd` or the member function `IloModel::add` to add this constraint to a model and then use `IloAlgorithm::extract` to extract the model for an algorithm in order for the constraint to be taken into account.

The arrays `cards` and `values` must be the same length; otherwise Concert Technology throws the exception `InvalidArraysException`.

public **IloDistribute**(const IloEnv env, const IloIntVarArray cards, const IloIntVarArray vars, const char * name=0)

This constructor creates a counting constraint in a model. You must use the template `IloAdd` or the member function `IloModel::add` to add this constraint to a model and then use `IloAlgorithm::extract` to extract the model for an algorithm in order for the constraint to be taken into account.

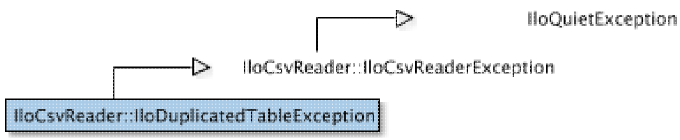
Methods

```
public IloDistributeI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

Class IloCsvReader::IloDuplicatedTableException

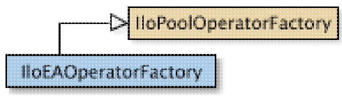
Definition file: ilconcert/ilocsvreader.h



Exception thrown for tables of same name in csv file.
This exception is thrown in the constructor of the csv reader if you read a multitable file in which two tables have the same name but table splitting has not been specified.

Class IloEAOperatorFactory

Definition file: ilsolver/iimarrayops.h
Include file: <ilsolver/iim.h>



A factory for generating operators over arrays of variables.
This factory provides a set of parameters as well as functions that build operators based on these parameters.

Assuming that you have already created:

- A solution prototype: (`prototype`).
- An array of decision variables (`vars`).

Then the following code creates two operators which operate on `vars`.

```
IloEAOperatorFactory factory(env, vars);
factory.setPrototype(prototype);
factory.setSearchLimit(IloFailLimit(env, 100));
IloPoolProc mut = factory.mutate(1.0 / vars.getSize(), "mutate");
IloPoolProc xo  = factory.uniformXover(0.5, "uXover");
```

Note
Due to the backtracking abilities of available operators, it is often necessary to set a search limit to avoid lengthy searches.

Constructor Summary	
`public`	`IloEAOperatorFactory(IloEAOperatorFactoryI * impl=0)` Initializes the handle.
`public`	`IloEAOperatorFactory(IloEnv env, IloIntVarArray scope)` Creates an operator factory.
`public`	`IloEAOperatorFactory(IloEnv env, IloBoolVarArray scope)` Creates an operator factory.

Method Summary	
`public IloEAOperatorFactoryI *`	`getImpl() const` Returns the pointer to the implementation class.
`public IloNum`	`getLeaveFactor() const` Returns the probability, for each variable, of being left unbound by created operators.
`public IloPoolOperator`	`mutate(IloNum mutationProb, const char * name=0) const` Creates a mutation operator.
`public IloPoolOperator`	`onePointXover(const char * name=0) const`

	Creates a one-point crossover operator.
public IloPoolOperator	randomize(const char * name=0) const Creates a randomization operator.
public IloPoolOperator	relax(const char * name=0) const Creates a uniform relaxation operator.
public IloPoolOperator	relaxSequence(IloNum breakProbability, const char * name=0) const Relaxes random parts of the array.
public void	setLeaveFactor(IloNum leaveFactor) const Specifies the probability, for each variable, of being left unbound by created operators.
public IloPoolOperator	swap(const char * name=0) const Creates a swap operator.
public IloPoolOperator	translocate(const char * name=0) const Creates a translocation operator.
public IloPoolOperator	transpose(const char * name=0) const Creates a transposition operator.
public IloPoolOperator	twoPointXover(const char * name=0) const Creates a two-point crossover operator.
public IloPoolOperator	uniformXover(const char * name=0) const Creates a uniform crossover operator with crossover rate 0.5.
public IloPoolOperator	uniformXover(IloNum xoverRate, const char * name=0) const Creates a uniform crossover operator.

Inherited Methods from `IloPoolOperatorFactory`
addAfterOperate, addBeforeOperate, addListener, end, getAfterOperate, getBeforeOperate, getEnv, getName, getObject, getPrototype, getSearchLimit, operator(), removeListener, setAfterOperate, setBeforeOperate, setName, setObject, setPrototype, setSearchLimit

Constructors

public **IloEAOperatorFactory**(IloEAOperatorFactoryI * impl=0)

Initializes the handle.

This constructor creates a handle object (an instance of the class `IloEAOperatorFactory`) from a pointer to an implementation object (an instance of the class `IloEAOperatorFactoryI`).

public **IloEAOperatorFactory**(IloEnv env, IloIntVarArray scope)

Creates an operator factory.

This constructor creates an operator factory associated with the environment `env` which will produce operators on the array of integer variables `scope`.

`public **IloEAOperatorFactory**(IloEnv env, IloBoolVarArray scope)`

Creates an operator factory.

This constructor creates an array operator factory associated with the environment `env` which will produce operators on the array of Boolean variables `scope`.

Methods

`public IloEAOperatorFactoryI * **getImpl**() const`

Returns the pointer to the implementation class.

This member function returns the implementation object of the invoking handle.

`public IloNum **getLeaveFactor**() const`

Returns the probability, for each variable, of being left unbound by created operators.

This member function returns the probability, for each array cell, of being left unbound by created operators. It returns the probability set at the previous call to `IloEAOperatorFactory::setLeaveFactor` or to 0.0 if no probability was set.

`public IloPoolOperator **mutate**(IloNum mutationProb, const char * name=0) const`

Creates a mutation operator.

This operator selects one parent and produces a solution by inheriting its values. Each inherited value is mutated with a given mutation probability `mutationProb`. `name`, if provided becomes the name of the newly created operator.

`public IloPoolOperator **onePointXover**(const char * name=0) const`

Creates a one-point crossover operator.

This recombination operator selects two parents and produces a new solution by inheriting values from the second parent for array indices greater than a randomly chosen cut point. Other values are inherited from the first parent. `name`, if provided becomes the name of the newly created operator.

`public IloPoolOperator **randomize**(const char * name=0) const`

Creates a randomization operator.

This operator instantiates variables by choosing random values in their domain. `name`, if provided becomes the name of the newly created operator.

`public IloPoolOperator **relax**(const char * name=0) const`

Creates a uniform relaxation operator.

This operator selects one parent and produces a new solution by leaving some variables unbound. The probability of leaving a variable unbound is given by the `leaveFactor` parameter of this factory. `name`, if provided becomes the name of the newly created operator.

See Also: IloEAOperatorFactory::setLeaveFactor, IloEAOperatorFactory::getLeaveFactor

`public IloPoolOperator` **`relaxSequence`**`(IloNum breakProbability, const char * name=0) const`

Relaxes random parts of the array.

This operator selects one parent and produces a new solution by copying the parent values with the exception of randomly chosen value sequences. The given probability `breakProbability` specifies, for each array site, the likelihood of starting (or ending) a relaxed sequence. `name`, if provided becomes the name of the newly created operator.

`public void` **`setLeaveFactor`**`(IloNum leaveFactor) const`

Specifies the probability, for each variable, of being left unbound by created operators.

This member function specifies the probability `leaveFactor`, for each cell, of being left unbound by created operators.

> **Note**
>
> You must specify a completion goal (with the `IloPoolOperatorFactory::setAfterOperate` or `IloPoolOperatorFactory::addAfterOperate` methods) if the `leaveFactor` parameter is non-zero. This goal should bind the variables that remain unbound.

`public IloPoolOperator` **`swap`**`(const char * name=0) const`

Creates a swap operator.

This swap operator selects a parent solution, chooses two random locations of the accessed array, and attempts to exchange their values. Upon failure, it backtracks to choose different exchange locations and fails when all exchange points have been tried. `name`, if provided becomes the name of the newly created operator.

`public IloPoolOperator` **`translocate`**`(const char * name=0) const`

Creates a translocation operator.

This translocation operator selects a parent solution and then chooses a random interval of the accessed array and a random insertion point. It then attempts to create a solution in which the sequence delimited by the chosen interval will be moved to the insertion point. Upon failure, it backtracks to choose a different insertion point. When this fails, it backtracks to change the interval to try to move. It fails when all intervals and insertion points have been tried. `name`, if provided becomes the name of the newly created operator.

`public IloPoolOperator` **`transpose`**`(const char * name=0) const`

Creates a transposition operator.

This transposition operator selects a parent solution and chooses a random interval of the accessed array. It then attempts to create a solution in which the sequence delimited by the chosen interval will be reversed. Upon failure, it backtracks to choose a different interval and fails when all intervals have been tried. `name`, if provided becomes the name of the newly created operator.

`public IloPoolOperator` **`twoPointXover`**`(const char * name=0) const`

Creates a two-point crossover operator.

This recombination operator selects two parents and produces a solution by inheriting values from the second parent for array indices contained between two randomly chosen cut points. Other values are inherited from the first parent. Upon failure, this goal backtracks by trying other cut points. `name`, if provided becomes the name of the newly created operator.

`public IloPoolOperator` **`uniformXover`**`(const char * name=0) const`

Creates a uniform crossover operator with crossover rate 0.5.

This recombination operator selects two parents and produces a new solution by inheriting values from the first or second parent based on a crossover rate of 0.5. This goal does not backtrack. `name`, if provided becomes the name of the newly created operator.

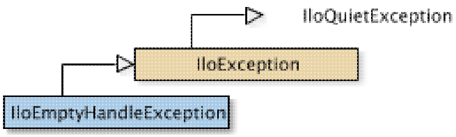
`public IloPoolOperator` **`uniformXover`**`(IloNum xoverRate, const char * name=0) const`

Creates a uniform crossover operator.

This recombination operator selects two parents and produces a new solution by inheriting values from the first or second parent based on the given crossover rate. This goal does not backtrack. The parameter `xoverRate` sets the probability of each feature being exchanged. `name`, if provided becomes the name of the newly created operator.

Class IloEmptyHandleException

Definition file: ilconcert/iloenv.h



The class of exceptions thrown if an empty handle is passed.
The exception `IloEmptyHandleException` is thrown if an empty handle is passed as an argument to a method, function, or class constructor.

Constructor and Destructor Summary	
`public`	`IloEmptyHandleException()`
`public`	`IloEmptyHandleException(const char * message)`

Inherited Methods from `IloException`
`end, getMessage`

Constructors and Destructors

`public` **`IloEmptyHandleException`**`()`

`public` **`IloEmptyHandleException`**`(const char * message)`

This constructor creates an exception containing the message string `message`.

Class IloEnv

Definition file: ilconcert/iloenv.h

The class of environments for models or algorithms in Concert Technology.
An instance of this class is an environment, managing memory and identifiers for modeling objects. Every Concert Technology object, such as an extractable object, a model, or an algorithm, must belong to an environment. In C++ terms, when you construct a model (an instance of `IloModel`) or an algorithm (an instance of `IloCplex`, `IloCP`, or `IloSolver`, for example), then you must pass one instance of `IloEnv` as an argument of that constructor.

Environment and Memory Management

An environment (an instance of `IloEnv`) efficiently manages memory allocations for the objects constructed with that environment as an argument. For example, when Concert Technology objects in your model are extracted by an algorithm, those extracted objects are handled as efficiently as possible with respect to memory management; there is no unnecessary copying that might cause memory explosions in your application on the part of Concert Technology.

When your application deletes an instance of `IloEnv`, Concert Technology will automatically delete all models and algorithms depending on that environment as well. You delete an environment by calling the member function `env.end`.

The memory allocated for Concert Technology arrays, expressions, sets, and columns is not freed until all references to these objects have terminated and the objects themselves have been deleted.

Certain classes documented in this manual, such as `IloFastMutex`, are known as system classes. They do not belong to a Concert Technology environment; in other words, an instance of `IloEnv` is *not* an argument in their constructors. As a consequence, a Concert Technology environment does *not* attempt to manage their memory allocation and de-allocation; a call of `IloEnv:end` will *not* delete an instance of a system class. These system classes are clearly designated in this documentation, and the appropriate constructors and destructors for them are documented in this manual as well.

Environment and Initialization

An instance of `IloEnv` in your application initializes certain data structures and modeling facilities for Concert Technology. For example, `IloEnv` initializes the symbolic constant `IloInfinity`.

The environment also specifies the current assumptions about normalization or the reduction of terms in linear expressions. For an explanation of this concept, see the concept Normalization: Reducing Linear Terms

Environment and Communication Streams

An instance of `IloEnv` in your application initializes the default output streams for general information, for error messages, and for warnings.

Environment and Extractable Objects

Every extractable object in your problem must belong to an instance of `IloEnv`. In C++ terms, in the constructor of certain extractable objects that you create, such as a constrained variable, you must pass an instance of `IloEnv` as an argument to specify which environment the extractable object belongs to. An extractable object (that is, an instance of `IloExtractable` or one of its derived subclasses) is tied throughout its lifetime to the environment where it is created. It can be used only with extractable objects belonging to the same environment. It can be extracted only for an algorithm attached to the same environment.

Two different environments cannot share the same extractable object.

You can extract objects from only one environment into a given algorithm. In other words, algorithms do not extract objects from two or more different environments.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloException, IloModel, operator new

Constructor Summary	
public	IloEnv()
public	IloEnv(IloEnvI * impl)

Method Summary	
public void	end()
public ostream &	error() const
public IloExtractableI *	getExtractable(IloInt id)
public IloEnvI *	getImpl() const
public IloInt	getMaxId() const
public IloInt	getMemoryUsage() const
public ostream &	getNullStream() const
public IloRandom	getRandom() const
public IloNum	getTime() const
public IloInt	getTotalMemoryUsage() const
public const char *	getVersion() const
public IloBool	isValidId(IloInt id) const
public ostream &	out() const
public void	printTime() const
public void	setDeleter(IloDeleterMode mode) const
public void	setError(ostream & s)
public void	setNormalizer(IloBool val) const
public void	setOut(ostream & s)
public void	setWarning(ostream & s)
public void	unsetDeleter() const
public ostream &	warning() const

Constructors

public **IloEnv**()

This constructor creates an environment to manage the extractable objects in Concert Technology.

public **IloEnv**(IloEnvI * impl)

This constructor creates an environment (a handle) from its implementation object.

Methods

`public void **end**()`

When you call this member function, it cleans up the invoking environment. In other words, it deletes all the extractable objects (instances of `IloExtractable` and its subclasses) created in that environment and frees the memory allocated for them. It also deletes all algorithms (instances of `IloAlgorithm` and its subclasses) created in that environment and frees memory allocated for them as well, including the representations of extractable objects extracted for those algorithms.

`public ostream & **error**() const`

This member function returns a reference to the output stream currently used for error messages from the invoking environment. It is initialized as `cerr`.

`public IloExtractableI * **getExtractable**(IloInt id)`

This member function returns the extractable associated with the specified identifier `id`.

`public IloEnvI * **getImpl**() const`

This member function returns the implementation object of the invoking environment.

`public IloInt **getMaxId**() const`

This member function returns the highest id of all extractables int the current `IloEnv`

`public IloInt **getMemoryUsage**() const`

This member function returns a value in bytes specifying how full the heap is.

`public ostream & **getNullStream**() const`

This member function calls the null stream of the environment. This member function can be used with `IloAlgorithm::setOut()` to suppress screen output by redirecting it to the null stream.

`public IloRandom **getRandom**() const`

Each instance of `IloEnv` contains a random number generator, an instance of the class `IloRandom`. This member function returns that `IloRandom` instance.

`public IloNum **getTime**() const`

This member function returns the amount of time elapsed in seconds since the construction of the invoking environment. (The member function `IloEnv::printTime` directs this information to the output stream of the invoking environment.)

`public IloInt` **`getTotalMemoryUsage`**`() const`

This member function returns a value in bytes specifying how large the heap is.

`public const char *` **`getVersion`**`() const`

This member function returns a string specifying the version of IBM® ILOG® Concert Technology.

`public IloBool` **`isValidId`**`(IloInt id) const`

This methods tells you if the current `id` is associated with a live extractable.

`public ostream &` **`out`**`() const`

This member function returns a reference to the output stream currently used for logging. General output from the invoking environment is accessible through this member function. By default, the logging output stream is defined by an instance of `IloEnv` as `cout`.

`public void` **`printTime`**`() const`

This member function directs the output of the member function `IloEnv::getTime` to the output stream of the invoking environment. (The member function `IloEnv::getTime` accesses the elapsed time in seconds since the creation of the invoking environment.)

`public void` **`setDeleter`**`(IloDeleterMode mode) const`

This member function sets the mode for the deletion of extractables, as described in the concept Deletion of Extractables. The mode can be `IloLinearDeleterMode` or `IloSafeDeleterMode`.

`public void` **`setError`**`(ostream & s)`

This member function sets the stream for errors generated by the invoking environment. By default, the stream is defined by an instance of `IloEnv` as `cerr`.

`public void` **`setNormalizer`**`(IloBool val) const`

This member function turns on or off the facilities in Concert Technology for normalizing linear expressions. Normalizing linear expressions is also known as reducing the terms of a linear expression. In this context, a linear expression that does not contain multiple terms with the same variable is said to be normalized. The concept in this manual offers examples of this idea.

When `val` is `IloTrue`, (the default), then Concert Technology analyzes linear expressions to determine whether any variable appears more than once in a given linear expression. It then combines terms in the linear expression to eliminate any duplication of variables. This mode may require more time during preliminary computation, but it avoids the possibility of an assertion failing in the case of duplicated variables in the terms of a linear expression.

When `val` is `IloFalse`, then Concert Technology assumes that all linear expressions in the invoking environment have already been processed to reduce them to their most efficient form. In other words, Concert Technology assumes that linear expressions have been normalized. This mode may save time during computation, but it entails the risk that a linear expression may contain one or more variables, each of which appears in one or more terms. This situation will cause certain `assert` statements in Concert Technology to fail if you do not compile with the flag `-DNDEBUG`.

`public void` **`setOut`**`(ostream & s)`

This member function redirects the `out()` stream with the stream given as an argument.

This member function can be used with `IloEnv::getNullStream` to suppress screen output by redirecting it to the null stream.

`public void` **`setWarning`**`(ostream & s)`

This member function sets the stream for warnings from the invoking environment. By default, the stream is defined by an instance of `IloEnv` as `cout`.

`public void` **`unsetDeleter`**`() const`

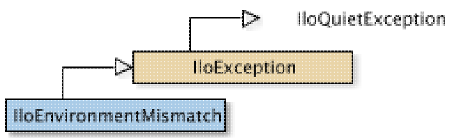
This member function unsets the mode for the deletion of extractables, as described in the concept Deletion of Extractables.

`public ostream &` **`warning`**`() const`

This member function returns a reference to the output stream currently used for warnings from the invoking environment. By default, the warning output stream is defined by an instance of `IloEnv` as `cout`.

Class IloEnvironmentMismatch

Definition file: ilconcert/iloenv.h



This exception is thrown if you try to build an object using objects from another environment.
The `IloEnvironmentMismatch` exception is thrown if you try to build an object using objects from another environment.

Constructor and Destructor Summary	
public	IloEnvironmentMismatch()
public	IloEnvironmentMismatch(const char * message)
public	~IloEnvironmentMismatch()

Inherited Methods from **IloException**
end, getMessage

Constructors and Destructors

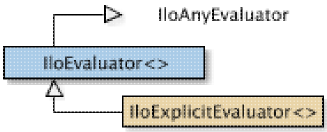
public **IloEnvironmentMismatch**()

public **IloEnvironmentMismatch**(const char * message)

public **~IloEnvironmentMismatch**()

Class IloEvaluator<>

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>



The `IloEvaluator` template class is the base class for all evaluators. An evaluator is a class that implements an evaluation on an object (using `operator()`) and returns an `IloNum` value. Such classes can be used (along with classes of `IloPredicate`) to parameterize selectors used in goal-based search.

Evaluators can easily be built by using the `ILOEVALUATOR0` macros or by composing existing evaluators and predicates.

For more information, see Selectors.

See Also: IloPredicate, IloTranslator, IloComparator

Method Summary	
`public IloComparator< IloObject >`	`makeGreaterThanComparator() const` Creates an evaluator-based comparator.
`public IloComparator< IloObject >`	`makeLessThanComparator() const` Creates an evaluator-based comparator.
`public IloNum`	`operator()(IloObject o, IloAny nu=0) const` Evaluates an object.

Methods

`public IloComparator< IloObject >` **`makeGreaterThanComparator`**`() const`

Creates an evaluator-based comparator.

This member function returns a comparator based on the invoking evaluator that prefers the objects with the highest value. More precisely, the comparator will return 1, 0, or -1 if the value of its left hand side is respectively greater than, equivalent, or less than the value of its right hand side.

`public IloComparator< IloObject >` **`makeLessThanComparator`**`() const`

Creates an evaluator-based comparator.

This member function returns a comparator based on the invoking evaluator that prefers the objects with the smallest value. More precisely, the comparator will return 1, 0, or -1 if the value of its left hand side is respectively less than, equivalent, or greater than the value of its right hand side.

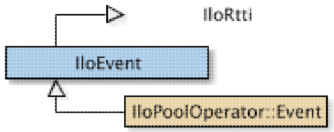
`public IloNum` **`operator()`**`(IloObject o, IloAny nu=0) const`

Evaluates an object.

This operator implements the evaluation of `o`, an instance of class `IloObject`, given a context `nu`. If the evaluator is a composite evaluator (see operators on evaluators), the context is also passed to the components (evaluators or predicates) of the composite evaluator.

Class IloEvent

Definition file: ilsolver/iimevent.h
Include file: <ilsolver/iim.h>

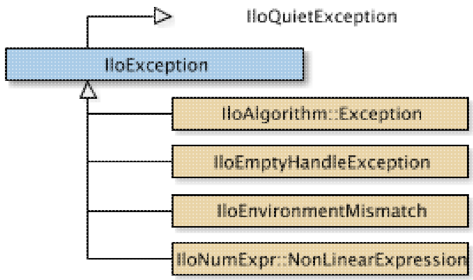


Basic event class.
Events are sent out by IIM (Iterative Improvement Methods) objects such as solution pools and pool operators. Normally, you will be interested in specific subclasses of events, and will create listeners to listen to those specific events.

See Also: IloListener, ILOIIMLISTENER0

Class IloException

Definition file: ilconcert/ilosys.h



Base class of Concert Technology exceptions.
This class is the base class for exceptions in Concert Technology. An instance of this class represents an exception on platforms that support exceptions when exceptions are enabled.

See Also: IloEnv, operator<<

Constructor and Destructor Summary	
`protected`	`IloException(const char * message=0, IloBool deleteMessage=IloFalse)`

Method Summary	
`public virtual void`	`end()`
`public virtual const char *`	`getMessage() const`

Constructors and Destructors

`protected` **`IloException`**`(const char * message=0, IloBool deleteMessage=IloFalse)`

This protected constructor creates an exception.

Methods

`public virtual void` **`end`**`()`

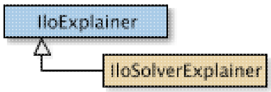
This member function deletes the invoking exception. That is, it frees memory associated with the invoking exception.

`public virtual const char *` **`getMessage`**`() const`

This member function returns the message (a character string) of the invoking exception.

Class IloExplainer

Definition file: ilsolver/iloexplain.h
Include file: <ilsolver/iloexplain.h>



Solver provides the ability to explain why a particular solution has been proposed. The class `IloExplainer` defines the abstract protocol of this explanation facility. This protocol consists of three queries:

- Why is a constraint `ct` satisfied?
- Why is a constraint `ct` violated?
- Why does a failure (that is, inconsistency) occur?

Subclasses of the class `IloExplainer` may use different deduction mechanisms for finding explanations. The subclass `IloSolverExplainer` uses the constraint propagation mechanism of Solver as a deduction mechanism.

> **Note**
>
> This functionality is available for pure Solver code only. It will not work on IBM® ILOG® Dispatcher or IBM ILOG Scheduler code.

Attribute Summary	
`protected IloExplainerI *`	`_impl`

Method Summary	
`public void`	`end()`
`public IloExplainerI *`	`getImpl() const`
`public IloModel`	`why(IloConstraint ct)`
`public IloModel`	`whyFail()`
`public IloModel`	`whyNot(IloConstraint ct)`

Attributes

`protected IloExplainerI *` **`_impl`**

This constructor creates a handle of an explanation object with the implementation object `impl`.

Methods

`public void` **`end`**`()`

This member function terminates the explainer and frees internally allocated memory.

`public IloExplainerI *` **`getImpl`**`() const`

This member function returns the implementation object of the invoking explanation object.

`public IloModel **why**(IloConstraint ct)`

If the constraint `ct` is satisfied in the solution of the invoking explainer, this member function returns an argument for `ct`. The argument is a minimal subset of the set of constraints of this solution that is sufficient to deduce the constraint `ct`. The argument is represented by an instance of the class `IloModel`. If the constraint `ct` is not satisfied in the solution of the invoking explainer, an exception `IloExplainer::NoExplanationException` is raised. If `ct` is an `IloExistsComponent` constraint, the argument shows why the associated component was included in the configuration.

`public IloModel **whyFail**()`

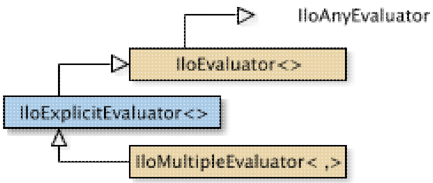
If the solution of the invoking explainer is inconsistent (that is, it leads to a failure when extracted by a solver), this member function returns a conflict. The conflict is a minimal subset of the set of constraints of this solution that is sufficient to produce a failure. The conflict is represented by an instance of the class `IloModel`. If the solution of the invoking explainer is consistent, an exception `IloExplainer::NoExplanationException` is raised.

`public IloModel **whyNot**(IloConstraint ct)`

If the constraint `ct` is violated in the solution of the invoking explainer, this member function returns a counterargument for `ct`. The counterargument is a minimal subset of the set of constraints of this solution that is sufficient to refute the constraint `ct`. The counterargument is represented by an instance of the class `IloModel`. If the constraint `ct` is not violated in the solution of the invoking explainer, an exception `IloExplainer::NoExplanationException` is raised.

Class IloExplicitEvaluator<>

Definition file: ilsolver/iimmulti.h
Include file: <ilsolver/iim.h>



An evaluator whose evaluations are specified explicitly.
There are various situations in which the evaluations on which some decisions are based do not change throughout a whole program or stage of a program. In this instance, it can be useful (for reasons of efficiency) to store object evaluations explicitly, rather than recalculate them each time. This class is a special type of evaluator which, when asked to evaluate an object, returns the evaluation from its store.

See Also: IloComparator, ILODEFAULTCOMPARATOR

Constructor Summary	
public	`IloExplicitEvaluator(IloEnv env, IloComparator< IloObject > cmp=0)` Creates an instance of an explicit evaluator.

Method Summary	
`public IloNum`	`getEvaluation(IloObject obj) const` Retrieves the evaluation of an object.
`public IloInt`	`getNumberOfEvaluations() const` Delivers the number of evaluations stored in the evaluator.
`public IloBool`	`hasEvaluation(IloObject obj) const` Determines of an object has an evaluation.
`public void`	`removeAllEvaluations()` Removes all evaluations from the evaluator.
`public void`	`removeEvaluation(IloObject obj)` Removes an evaluation from the evaluator.
`public void`	`setEvaluation(IloObject obj, IloNum eval)` Sets the evaluation of an object.

Inherited Methods from `IloEvaluator`
`makeGreaterThanComparator, makeLessThanComparator, operator()`

Inner Class	
IloExplicitEvaluator::Iterator	An iterator which will iterate over all evaluated objects in an explicit evaluator.

Constructors

`public IloExplicitEvaluator(IloEnv env, IloComparator< IloObject > cmp=0)`

Creates an instance of an explicit evaluator.

This constructor creates an instance of an explicit evaluator from an environment `env` and an optional comparator `cmp`. In order to correctly retrieve object evaluations, this class needs to know how to compare the objects being evaluated, so that when you ask for an evaluation of an object, the evaluator can look up this object in its store. To do this, you may pass a comparator `cmp` which can make this comparison. Note that this comparator is not used to determine if one object is more favorable than another (for example see `IloBestSolutionComparator`), but should compare the objects themselves.

If no comparator is passed, the evaluator tries to find one by looking for those defined using `ILODEFAULTCOMPARATOR`. If such a comparator has been defined, it will be used for object comparison. The Solver IIM library has default comparators pre-defined for classes `IloSolution` and `IloPoolProc`. The code for the default comparator for an `IloPoolOperator` could be defined as follows:

```
ILODEFAULTCOMPARATOR(IloPoolOperator, a, b) {
  return a.getImpl() < b.getImpl();
}
```

See Also: IloComparator, IloSolution, IloPoolProc, IloBestSolutionComparator, ILODEFAULTCOMPARATOR

Methods

`public IloNum getEvaluation(IloObject obj) const`

Retrieves the evaluation of an object.

This member function retieves the valuation of an object `obj` previously set using `IloExplicitEvaluator::setEvaluation`. If the object in question has no evaluation associated with it in the invoking object, and exception (of type `IloException`) is raised.

> **Note**
>
> This function exists mostly for symmetry with the `IloExplicitEvaluator::setEvaluation` member function. Normally the evaluation of an object will be retieved via the paranthesis operator on the evaluator. That is, for an evaluator of type `IloExplicitEvaluator`, `evaluator.getEvaluation(obj)` and `evaluator(obj)` are equivalent.

`public IloInt getNumberOfEvaluations() const`

Delivers the number of evaluations stored in the evaluator.

This member function returns the number of evaluations which have been added to the evaluator. It can be useful for dimensioning arrays and so forth before iterating over the evaluations.

`public IloBool hasEvaluation(IloObject obj) const`

Determines of an object has an evaluation.

This member function determines if a given object `obj` has had an evaluation set via `IloExplicitEvaluator::setEvaluation`.

`public void removeAllEvaluations()`

Removes all evaluations from the evaluator.

The member function removes all evaluations from the evaluation, and is particularly useful when you wish to start again with an empty evaluator.

`public void` **`removeEvaluation`**`(IloObject obj)`

Removes an evaluation from the evaluator.

The member function removes the evaluation of a particular object `obj` from the evaluator. This does not mean that the evaluation is set to a particular value (for example zero), but that after the call, the invoking evaluator will have no evaluation at all for the specified object.

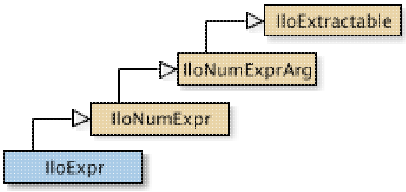
`public void` **`setEvaluation`**`(IloObject obj, IloNum eval)`

Sets the evaluation of an object.

This member function is used to set the evaluation of an object `obj` to value `eval`. If the given object has no evaluation in the invoking evaluation, that object is added together with its new evaluation. Otherwise, the evaluation of the specified object is changed to the newly specified value.

Class IloExpr

Definition file: ilconcert/iloexpression.h



An instance of this class represents an expression in a model.
An instance of `IloExpr` is a handle.

Expressions in Environments

The variables in an expression must all belong to the same environment as the expression itself. In other words, you must not mix variables from different environments within the same expression.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Programming Hint: Creating Expressions

In addition to using a constructor of this class to create an expression, you may also initialize an instance of `IloExpr` as a C++ expression built from variables of a model. For example:

```
IloNumVar x;
IloNumVar y;
IloExpr expr = x + y;
```

Programming Hint: Empty Handles and Null Expressions

This statement creates an empty handle:

```
IloExpr e1;
```

You must initialize it before you use it. For example, if you attempt to use it in this way:

```
e1 += 10; // BAD IDEA
```

Without the compiler option `-DNDEBUG`, that line will cause an `assert` statement to fail because you are attempting to use an empty handle.

In contrast, the following statement

```
IloExpr e2(env);
```

creates a handle to a null expression. You can use this handle to build up an expression, for example, in this way:

```
e2 += 10; // OK
```

Normalizing Linear Expressions: Reducing the Terms

Normalizing is sometimes known as *reducing the terms* of a linear expression.

Linear expressions consist of terms made up of constants and variables related by arithmetic operations; for example, x + 3y is a linear expression of two terms consisting of two variables. In some expressions, a given variable may appear in more than one term, for example, x + 3y +2x. Concert Technology has more than one way of dealing with linear expressions in this respect, and you control which way Concert Technology treats expressions from your application.

In one mode, Concert Technology analyzes linear expressions that your application passes it and attempts to reduce them so that a given variable appears in only one term in the linear expression. This is the default mode. You set this mode with the member function `IloEnv::setNormalizer(IloTrue).`

In the other mode, Concert Technology assumes that no variable appears in more than one term in any of the linear expressions that your application passes to Concert Technology. We call this mode assume normalized linear expressions. You set this mode with the member function `IloEnv::setNormalizer(IloFalse).`

Certain constructors and member functions in this class check this setting in the environment and behave accordingly: they assume that no variable appears in more than one term in a linear expression. This mode may save time during computation, but it entails the risk that a linear expression may contain one or more variables, each of which appears in one or more terms. Such a case may cause certain assertions in member functions of this class to fail if you do not compile with the flag `-DNDEBUG`.

Certain constructors and member functions in this class check this setting in the environment and behave accordingly: they attempt to reduce expressions. This mode may require more time during preliminary computation, but it avoids of the possibility of a failed assertion in case of duplicates.

See Also: IloExprArray, IloModel

Constructor Summary	
public	IloExpr()
public	IloExpr(IloNumExprI * expr)
public	IloExpr(const IloNumLinExprTerm term)
public	IloExpr(const IloIntLinExprTerm term)
public	IloExpr(IloNumExprArg)
public	IloExpr(const IloEnv env, IloNum=0)

Method Summary	
public IloNum	getConstant() const
public IloNumLinTermI *	getImpl() const
public IloExpr::LinearIterator	getLinearIterator() const
public IloBool	isNormalized() const
public IloInt	normalize() const
public IloExpr &	operator*=(IloNum val)
public IloExpr &	operator+=(const IloIntLinExprTerm term)
public IloExpr &	operator+=(const IloNumLinExprTerm term)
public IloExpr &	operator+=(const IloIntVar var)
public IloExpr &	operator+=(const IloNumVar var)
public IloExpr &	operator+=(const IloNumExprArg expr)
public IloExpr &	operator+=(IloNum val)
public IloExpr &	operator-=(const IloIntLinExprTerm term)
public IloExpr &	operator-=(const IloNumLinExprTerm term)
public IloExpr &	operator-=(const IloIntVar var)
public IloExpr &	operator-=(const IloNumVar var)
public IloExpr &	operator-=(const IloNumExprArg expr)

public IloExpr &	operator-=(IloNum val)
public IloExpr &	operator/=(IloNum val)
public void	remove(const IloNumVarArray vars)
public void	setConstant(IloNum cst)
public void	setLinearCoef(const IloNumVar var, IloNum value)
public void	setLinearCoefs(const IloNumVarArray vars, IloNumArray values)
public void	setNumConstant(IloNum constant)

Inherited Methods from `IloNumExpr`
getImpl, operator*=, operator+=, operator+=, operator-=, operator-=, operator/=

Inherited Methods from `IloNumExprArg`
getImpl

Inherited Methods from `IloExtractable`
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Inner Class	
IloExpr::LinearIterator	An iterator over the linear part of an expression.

Constructors

public **IloExpr**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IloExpr**(IloNumExprI * expr)

This constructor creates an expression from a pointer to the implementation class of numeric expressions `IloNumExprI*`.

public **IloExpr**(const IloNumLinExprTerm term)

This constructor creates an integer expression with linear terms using the undocumented class `IloNumLinExprTerm`.

public **IloExpr**(const IloIntLinExprTerm term)

This constructor creates an integer expression with linear terms using the undocumented class `IloIntLinExprTerm`.

public **IloExpr**(IloNumExprArg)

This constructor creates an expression using the undocumented class `IloNumExprArg`.

```
public IloExpr(const IloEnv env, IloNum=0)
```

This constructor creates an expression in the environment specified by `env`. It may be used to build other expressions from variables belonging to `env`. You must not mix variables of different environments within an expression.

Methods

```
public IloNum getConstant() const
```

This member function returns the constant term in the invoking expression.

```
public IloNumLinTermI * getImpl() const
```

This member function returns the implementation object of the invoking enumerated variable.

```
public IloExpr::LinearIterator getLinearIterator() const
```

This methods returns a linear iterator on the invoking expression.

```
public IloBool isNormalized() const
```

This member function returns `IloTrue` if the invoking expression has been normalized using `IloExpr::normalize`.

```
public IloInt normalize() const
```

This member function normalizes the invoking linear expression. Normalizing is sometimes known as reducing the terms of a linear expression. That is, if there is more than one linear term using the same variable in the invoking linear expression, then this member function merges those linear terms into a single term expressed in that variable. The return value specifies the number of merged terms.

For example, `1*x + 17*y - 3*x` becomes `17*y - 2*x`, and the member function returns 1 (one).

If you attempt to use this member function on a nonlinear expression, it throws an exception.

```
public IloExpr & operator*=(IloNum val)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x * ...`

```
public IloExpr & operator+=(const IloIntLinExprTerm term)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x + ...`

`public IloExpr & **operator+=**(const IloNumLinExprTerm term)`

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x + ...`

`public IloExpr & **operator+=**(const IloIntVar var)`

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x + ...`

`public IloExpr & **operator+=**(const IloNumVar var)`

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x + ...`

`public IloExpr & **operator+=**(const IloNumExprArg expr)`

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x + ...`

`public IloExpr & **operator+=**(IloNum val)`

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x + ...`

`public IloExpr & **operator-=**(const IloIntLinExprTerm term)`

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x - ...`

`public IloExpr & **operator-=**(const IloNumLinExprTerm term)`

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x - ...`

`public IloExpr & **operator-=**(const IloIntVar var)`

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x - ...`

`public IloExpr & **operator-=**(const IloNumVar var)`

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x - ...`

```
public IloExpr & operator-=(const IloNumExprArg expr)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x - ...`

```
public IloExpr & operator-=(IloNum val)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x - ...`

```
public IloExpr & operator/=(IloNum val)
```

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x / ...`

```
public void remove(const IloNumVarArray vars)
```

This member function removes all occurrences of all variables listed in the array `vars` from the invoking expression. For linear expressions, the effect of this member function is equivalent to setting the coefficient for all the variables listed in `vars` to 0 (zero).

```
public void setConstant(IloNum cst)
```

This member function assigns `cst` as the constant term in the invoking expression.

```
public void setLinearCoef(const IloNumVar var, IloNum value)
```

This member function assigns `value` as the coefficient of `var` in the invoking expression if the invoking expression is linear. This member function applies only to linear expressions. In other words, you can not use this member function to change the coefficient of a non linear expression. An attempt to do so will cause Concert Technology to throw an exception.

```
public void setLinearCoefs(const IloNumVarArray vars, IloNumArray values)
```

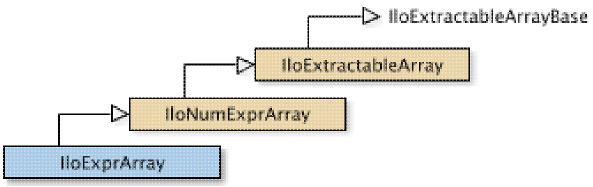
For each of the variables in `vars`, this member function assigns the corresponding value of `values` as its linear coefficient if the invoking expression is linear. This member function applies only to linear expressions. In other words, you can not use this member function to change the coefficient of a nonlinear expression. An attempt to do so will cause Concert Technology to throw an exception.

```
public void setNumConstant(IloNum constant)
```

This member function assigns `cst` as the constant term in the invoking expression.

Class IloExprArray

Definition file: ilconcert/iloexpression.h



The array class of the expressions class.
For each basic type, Concert Technology defines a corresponding array class. `IloExprArray` is the array class of the expressions class (`IloExpr`) for a model.

Instances of `IloExprArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloExpr

Constructor Summary	
`public`	`IloExprArray(IloDefaultArrayI * i=0)`
`public`	`IloExprArray(const IloEnv env, IloInt n=0)`

Method Summary	
`public IloNumExprArg`	`operator[](IloIntExprArg anIntegerExpr) const`

Inherited Methods from `IloNumExprArray`
`add, add, add, endElements, operator[]`

Inherited Methods from `IloExtractableArray`
`add, add, add, endElements, setNames`

Constructors

`public` **`IloExprArray`**`(IloDefaultArrayI * i=0)`

This constructor creates an empty array of expressions for use in a model. You cannot create instances of the undocumented class `IloDefaultArrayI`. As an argument in this default constructor, it allows you to pass 0 (zero) as a value to an optional argument in functions and member functions that accept an array as an argument.

`public` **`IloExprArray`**`(const IloEnv env, IloInt n=0)`

This constructor creates an array of `n` elements. Initially, the `n` elements are empty handles.

Methods

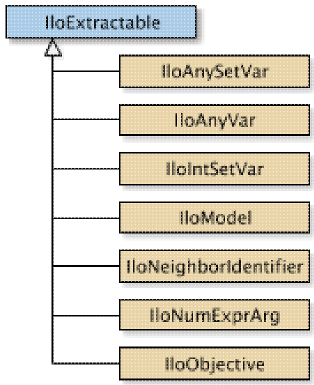
`public IloNumExprArg operator[](IloIntExprArg anIntegerExpr) const`

This subscripting operator returns an expression argument for use in a constraint or expression. For clarity, let's call `A` the invoking array. When `anIntegerExpr` is bound to the value `i`, the domain of the expression is the domain of `A[i]`. More generally, the domain of the expression is the union of the domains of the expressions `A[i]` where the `i` are in the domain of `anIntegerExpr`.

This operator is also known as an element expression.

Class IloExtractable

Definition file: ilconcert/iloextractable.h



Base class of all extractable objects.
This class is the base class of all extractable objects (that is, instances of such classes as `IloConstraint`, `IloNumVar`, and so forth). Instances of subclasses of this class represent objects (such as constraints, constrained variables, objectives, and so forth) that can be extracted by Concert Technology from your model for use by your application in Concert Technology algorithms.

Not every algorithm can extract every extractable object of a model. For example, a model may include more than one objective, but you can extract only one objective for an instance of `IloCplex`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Adding Extractable Objects

Generally, for an extractable object to be taken into account by one of the algorithms in Concert Technology, you must add the extractable object to a model with the member function `IloModel::add` and extract the model for the algorithm with the member function `IloAlgorithm::extract`.

Environment and Extractable Objects

Every extractable object in your model must belong to one instance of `IloEnv`. An extractable object (that is, an instance of `IloExtractable` or one of its derived subclasses) is tied throughout its lifetime to the environment where it is created. It can be used only with extractable objects belonging to the same environment. It can be extracted only for an algorithm attached to the same environment.

Notification

When you change an extractable object, for example by removing it from a model, Concert Technology notifies algorithms that have extracted the model containing this extractable object about the change. Member functions that carry out such notification are noted in this documentation.

See Also: IloEnv, IloGetClone, IloModel

Constructor Summary	
public	`IloExtractable(IloExtractableI * obj=0)`

Method Summary	
public IloConstraint	`asConstraint() const`
public IloIntExprArg	`asIntExpr() const`

public IloModel	asModel() const
public IloNumExprArg	asNumExpr() const
public IloObjective	asObjective() const
public IloNumVar	asVariable() const
public void	end()
public IloEnv	getEnv() const
public IloInt	getId() const
public IloExtractableI *	getImpl() const
public const char *	getName() const
public IloAny	getObject() const
public IloBool	isConstraint() const
public IloBool	isIntExpr() const
public IloBool	isModel() const
public IloBool	isNumExpr() const
public IloBool	isObjective() const
public IloBool	isVariable() const
public void	setName(const char * name) const
public void	setObject(IloAny obj) const

Constructors

public **IloExtractable**(IloExtractableI * obj=0)

This constructor creates a handle to the implementation object.

Methods

public IloConstraint **asConstraint**() const

This method returns the given extractable as a constraint or a null pointer

See IloExtractableVisitor if you want to introspect an expression

See Also: IloExtractableVisitor

public IloIntExprArg **asIntExpr**() const

This method returns the given extractable as an integer expression or a null pointer

See IloExtractableVisitor if you want to introspect an expression

See Also: IloExtractableVisitor

public IloModel **asModel**() const

This method returns the given extractable as a model or a null pointer

See IloExtractableVisitor if you want to introspect an expression

See Also: IloExtractableVisitor

```
public IloNumExprArg asNumExpr() const
```

This method returns the given extractable as a floating expression or a null pointer

See IloExtractableVisitor if you want to introspect an expression

See Also: IloExtractableVisitor

```
public IloObjective asObjective() const
```

This method returns the given extractable as an objective or a null pointer

See IloExtractableVisitor if you want to introspect an expression

See Also: IloExtractableVisitor

```
public IloNumVar asVariable() const
```

This method returns the given extractable as a variable or a null pointer

See IloExtractableVisitor if you want to introspect an expression

See Also: IloExtractableVisitor

```
public void end()
```

This member function first removes the invoking extractable object from all other extractable objects where it is used (such as a model, ranges, etc.) and then deletes the invoking extractable object. That is, it frees all the resources used by the invoking object. After a call to this member function, you can not use the invoking extractable object again.

Note
The member function `end` notifies Concert Technology algorithms about the destruction of this invoking object.

```
public IloEnv getEnv() const
```

This member function returns the environment to which the invoking extractable object belongs. An extractable object belongs to exactly one environment; different environments can not share the same extractable object.

```
public IloInt getId() const
```

This member function returns the ID of the invoking extractable object.

`public IloExtractableI * getImpl() const`

This member function returns a pointer to the implementation object of the invoking extractable object. This member function is useful when you need to be sure that you are using the same copy of the invoking extractable object in more than one situation.

`public const char * getName() const`

This member function returns a character string specifying the name of the invoking object (if there is one).

`public IloAny getObject() const`

This member function returns the object associated with the invoking object (if there is one). Normally, an associated object contains user data pertinent to the invoking object.

`public IloBool isConstraint() const`

This method tells you whether or not the given extractable is a constraint or not

See IloExtractableVisitor if you want to introspect an expression

See Also: IloExtractableVisitor

`public IloBool isIntExpr() const`

This method tells you whether or not the given extractable is an integer expression or not

See IloExtractableVisitor if you want to introspect an expression

See Also: IloExtractableVisitor

`public IloBool isModel() const`

This method tells you whether ot not the given extractable is a model or not

See IloExtractableVisitor if you want to introspect an expression

See Also: IloExtractableVisitor

`public IloBool isNumExpr() const`

This method tells you whether or not the given extractable is a floating expression or not

See IloExtractableVisitor if you want to introspect an expression

See Also: IloExtractableVisitor

`public IloBool **isObjective**() const`

This method tells you whether or not the given extractable is an objective or not

See IloExtractableVisitor if you want to introspect an expression

See Also: IloExtractableVisitor

`public IloBool **isVariable**() const`

This method tells you whether or not the given extractable is a variable or not

See IloExtractableVisitor if you want to introspect an expression

See Also: IloExtractableVisitor

`public void **setName**(const char * name) const`

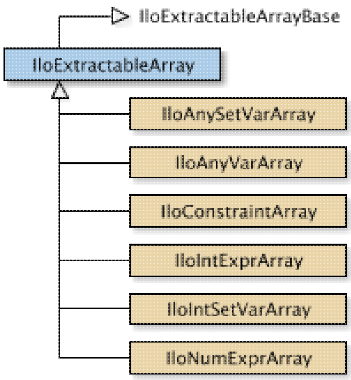
This member function assigns `name` to the invoking object.

`public void **setObject**(IloAny obj) const`

This member function associates `obj` with the invoking object. The member function `getObject` accesses this associated object afterward. Normally, `obj` contains user data pertinent to the invoking object.

Class IloExtractableArray

Definition file: ilconcert/iloextractable.h



An array of extractable objects.
An instance of this class is an array of extractable objects (instances of the class `IloExtractable` or its subclasses).

Instances of `IloExtractableArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

For information on arrays, see the concept Arrays

See Also: IloArray, IloExtractable, operator<<

Constructor Summary	
`public`	`IloExtractableArray(IloDefaultArrayI * i=0)`
`public`	`IloExtractableArray(const IloExtractableArray & r)`
`public`	`IloExtractableArray(const IloEnv env, IloInt n=0)`

Method Summary	
`public void`	`add(IloInt more, const IloExtractable x)`
`public void`	`add(const IloExtractable x)`
`public void`	`add(const IloExtractableArray x)`
`public void`	`endElements()`
`public void`	`setNames(const char * name)`

Constructors

`public` **`IloExtractableArray`**`(IloDefaultArrayI * i=0)`

This constructor creates an empty array of elements. You cannot create instances of the undocumented class `IloDefaultArrayI`. As an argument in this default constructor, it allows you to pass 0 (zero) as a value to an optional argument in functions and member functions that accept an array as an argument.

`public IloExtractableArray(const IloExtractableArray & r)`

This copy constructor creates a handle to the array of extractable objects specified by `r`.

`public IloExtractableArray(const IloEnv env, IloInt n=0)`

This constructor creates an array of `n` elements, each of which is an empty handle.

Methods

`public void add(IloInt more, const IloExtractable x)`

This member function appends `x` to the invoking array multiple times. The argument `more` specifies how many times.

`public void add(const IloExtractable x)`

This member function appends `x` to the invoking array.

`public void add(const IloExtractableArray x)`

This member function appends the elements in the array`x` to the invoking array.

`public void endElements()`

This member function calls `IloExtractable::end` for each of the elements in the invoking array. This deletes all the extractables identified by the array, leaving the handles in the array intact. This member function is the recommended way to delete the elements of an array.

`public void setNames(const char * name)`

This member function set the name for all elements of the invoking array. All elements must be different, otherwise raise an error.

Class IloExtractableVisitor

Definition file: ilconcert/iloextractable.h

The class for inspecting all nodes of an expression.
The class `IloExtractableVisitor` is used to introspect a Concert object and inspect all nodes of the expression.

For example, you can introspect a given expression and look for all the variables within.

You can do this by specializing the `visitChildren` methods and calling the `beginVisit` method on the extractable you want to introspect.

For example, if you visit an `IloDiff` object, you will visit the first expression, then the second expression. When visiting the first or second expression, you will visit their sub-expressions, and so on.

Constructor and Destructor Summary	
`public`	`IloExtractableVisitor()`

Method Summary	
`public virtual void`	`beginVisit(IloExtractableI * e)`
`public virtual void`	`endVisit(IloExtractableI * e)`
`public virtual void`	`visitChildren(IloExtractableI * parent, IloExtractableArray children)`
`public virtual void`	`visitChildren(IloExtractableI * parent, IloExtractableI * child)`

Constructors and Destructors

`public` **`IloExtractableVisitor`**`()`

The default constructor.

Methods

`public virtual void` **`beginVisit`**`(IloExtractableI * e)`

This method begins the introspection.

`public virtual void` **`endVisit`**`(IloExtractableI * e)`

This method ends the inspection.

`public virtual void` **`visitChildren`**`(IloExtractableI * parent, IloExtractableArray children)`

This method is called when the member of the object is an array.

For example, when visiting an `IloAllDiff(env, [x,y,z])` , you use

```
visitChildren(AllDiff, [x,y,z])
```

`public virtual void` **`visitChildren`**`(IloExtractableI * parent, IloExtractableI * child)`

This method is called when visiting a sub-extractable.

For example, if you want to display all the variables in your object, you use:

```
visitChildren(IloExtractableI* parent, IloExtractableI* child){
IloExtractable extr(child); if (child.isVariable()) cout << extr;
}
```

If you visit `IloDiff(env, X, Y)` , for example, you would call this method as:

```
visitChildren(Diff, X)
```

then

```
visitChildren(Diff, Y)
```

Class IloFastMutex

Definition file: ilconcert/ilothread.h

Synchronization primitives adapted to the needs of Concert Technology.
The class `IloFastMutex` provides synchronization primitives adapted to the needs of Concert Technology. In particular, an instance of the class `IloFastMutex` is a nonrecursive mutex that implements mutual exclusion from critical sections of code in multithreaded applications. The purpose of a mutex is to guarantee that concurrent calls to a critical section of code in a multithreaded application are serialized. If a critical section of code is protected by a mutex, then two (or more) threads cannot execute the critical section simultaneously. That is, an instance of this class makes it possible for you to serialize potentially concurrent calls.

Concert Technology implements a mutex by using a single resource that you lock when your application enters the critical section and that you unlock when you leave. Only one thread can own that resource at a given time.

Protection by a Mutex

A critical section of code in a multithreaded application is protected by a mutex when that section of code is encapsulated by a pair of calls to the member functions `IloFastMutex::lock` and `IloFastMutex::unlock`.

In fact, we say that a pair of calls to the member functions `lock` and `unlock` defines a critical section. The conventional way of defining a critical section looks like this:

```
mutex.lock();
while (conditionC does not hold)
       condition.wait(&mutex);
doTreatmentT();
mutex.unlock();
```

The class `IloCondition` provides synchronization primitives to express conditions in critical sections of code.

State of a Mutex

A mutex (an instance of `IloFastMutex`) has a state; the state may be locked or unlocked. You can inquire about the state of a mutex to determine whether it is locked or unlocked by using the member function `IloFastMutex::isLocked`. When a thread enters a critical section of code in a multithreaded application and then locks the mutex defining that critical section, we say that the thread owns that lock and that lock belongs to the thread until the thread unlocks the mutex.

Exceptions

The member functions `IloFastMutex::lock` and `IloFastMutex::unlock` can throw C++ exceptions when exceptions are enabled on platforms that support them. These are the possible exceptions:

- `IloMutexDeadlock`: Instances of `IloFastMutex` are not recursive. Consequently, if a thread locks a mutex and then attempts to lock that mutex again, the member function `lock` throws the exception `MutexDeadlock`. On platforms that do not support exceptions, it causes the application to exit.
- `IloMutexNotOwner`: The thread that releases a given lock (that is, the thread that unlocks a mutex) must be the same thread that locked the mutex in the first place. For example, if a thread `A` takes lock `L` and thread `B` attempts to unlock `L`, then the member function `unlock` throws the exception `MutexNotOwner`. On platforms that do not support exceptions, it causes the application to exit.
- `IloMutexNotOwner`: The member function `unlock` throws this exception whenever a thread attempts to unlock an instance of `IloFastMutex` that is not already locked. On platforms that do not support exceptions, it causes the application to exit.

System Class: Memory Management

`IloFastMutex` is a system class.

Most Concert Technology classes are actually handle classes whose instances point to objects of a corresponding implementation class. For example, instances of the Concert Technology class `IloNumVar` are handles pointing to instances of the implementation class `IloNumVarI`. Their allocation and de-allocation in internal data structures of Concert Technology are managed by an instance of `IloEnv`.

However, system classes, such as `IloFastMutex`, differ from that pattern. `IloFastMutex` is an ordinary C++ class. Its instances are allocated on the C++ heap.

Instances of `IloFastMutex` are not automatically de-allocated by a call to `IloEnv::end`. You must explicitly destroy instances of `IloFastMutex` by means of a call to the delete operator (which calls the appropriate destructor) when your application no longer needs instances of this class.

Furthermore, you should not allocate—neither directly nor indirectly—any instance of `IloFastMutex` in the Concert Technology environment because the destructor for that instance of `IloFastMutex` will never be called automatically by `IloEnv::end` when it cleans up other Concert Technology objects in the Concert Technology environment. In other words, allocation of any instance of `IloFastMutex` in the Concert Technology environment will produce memory leaks.

For example, it is not a good idea to make an instance of `IloFastMutex` part of a conventional Concert Technology model allocated in the Concert Technology environment because that instance will not automatically be de-allocated from the Concert Technology environment along with the other Concert Technology objects.

De-allocating Instances of IloFastMutex

Instances of `IloFastMutex` differ from the usual Concert Technology objects because they are not allocated in the Concert Technology environment, and their de-allocation is not managed automatically for you by `IloEnv::end`. Instead, you must explicitly destroy instances of `IloFastMutex` by calling the delete operator when your application no longer needs those objects.

See Also: IloBarrier, IloCondition

Constructor and Destructor Summary	
`public`	`IloFastMutex()`
`public`	`~IloFastMutex()`

Method Summary	
`public int`	`isLocked()`
`public void`	`lock()`
`public void`	`unlock()`

Constructors and Destructors

`public` **`IloFastMutex`**`()`

This constructor creates an instance of `IloFastMutex` and allocates it on the C++ heap (not in the Concert Technology environment). This mutex contains operating system-specific resources to represent a lock. You may use this mutex for purposes that are private to a process. Its behavior is undefined for inter-process locking.

`public` **`~IloFastMutex`**`()`

The delete operator calls this destructor to de-allocate an instance of `IloFastMutex`. This destructor is called automatically by the runtime system. The destructor releases operating system-specific resources of the invoking mutex.

Methods

`public int isLocked()`

This member function returns a Boolean value that shows the state of the invoking mutex. That is, it tells you whether the mutex is locked by the calling thread (`0`) or unlocked (`1`) or locked by a thread other than the calling thread (also `1`).

`public void lock()`

This member function acquires a lock for the invoking mutex on behalf of the calling thread. That lock belongs to the calling thread until the member function `unlock` is called.

If you call this member function and the invoking mutex has already been locked, then the calling thread is suspended until the first lock is released.

`public void unlock()`

This member function releases the lock on the invoking mutex, if there is such a lock.

If you call this member function on a mutex that has not been locked, then this member function throws an exception if C++ exceptions have been enabled on a platform that supports exceptions. Otherwise, it causes the application to exit.

Class IloCsvReader::IloFieldNotFoundException

Definition file: ilconcert/ilocsvreader.h

Exception thrown for field not found.
This exception is thrown by the `IloCsvLine` methods listed below if the corresponding field does not exist.

- IloCsvLine::getFloatByPosition
- IloCsvLine::getIntByPosition
- IloCsvLine::getStringByPosition
- IloCsvLine::getFloatByHeader
- IloCsvLine::getIntByHeader
- IloCsvLine::getStringByHeader
- IloCsvLine::getFloatByPositionOrDefaultValue
- IloCsvLine::getIntByPositionOrDefaultValue
- IloCsvLine::getStringByPositionOrDefaultValue
- IloCsvLine::getFloatByHeaderOrDefaultValue
- IloCsvLine::getIntByHeaderOrDefaultValue
- IloCsvLine::getStringByHeaderOrDefaultValue

Class IloCsvReader::IloFileNotFoundException

Definition file: ilconcert/ilocsvreader.h

Exception thrown when file is not found.
This exception is thrown in the constructor of the csv reader if a specified file is not found.

Class IloFunction<,>

Definition file: ilconcert/iloset.h

For constraint programming: A template for creating a handle class to the implementation class built by the template `IloFunctionI`.
Concert Technology offers you the means to define classes of functions that map instances of one class `X` to instances of another class `Y`.

This C++ template creates a class of handles to the implementation class built by the template `IloFunctionI`.

Normally, you subclass the class `IloFunctionI<X,Y>` and in doing so, you define its pure virtual member function:

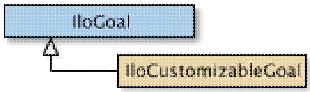
```
virtual  Y getValue(X);
```

Then you use this template to define a handle to that class of type `IloFunction<X,Y>.`

Definition file:<ilconcert/iloset.h>

Class IloGoal

Definition file: ilsolver/ilosolverhandle.h
Include file: <ilsolver/ilosolver.h>



An instance of `IloGoal` represents a goal, a search primitive, for use in a Concert Technology model. That is, a goal is a building block in a search for solutions.

The goals listed in *See Also* are predefined in IBM® ILOG® Solver.

Environment and Goals

Every instance of `IloGoal` must belong to one instance of `IloEnv`. A goal (that is, an instance of `IloGoal` or one of its derived subclasses) is tied throughout its lifetime to the environment where it is created. It can be used only with goals and extractable objects belonging to the same environment.

See Also: IloAndGoal, IloApply, IloBestGenerate, IloBestInstantiate, IloDichotomize, IloGenerate, IloGoalFail, IloGoalTrue, IloInstantiate, IloLimitSearch, IloOrGoal, IloRemoveValue, IloSetMax, IloSetMin, IloSetValue, ILOCPGOALWRAPPER0

Constructor Summary	
`public`	`IloGoal()`
`public`	`IloGoal(IloGoalI * impl=0)`

Method Summary	
`public void`	`end() const`
`public IloEnv`	`getEnv() const`
`public IloGoalI *`	`getImpl() const`
`public const char *`	`getName() const`
`public IloAny`	`getObject() const`
`public void`	`setName(const char * name) const`
`public void`	`setObject(IloAny obj) const`

Constructors

`public` **`IloGoal`**`()`

This constructor creates a goal which is empty, that is, one whose handle pointer is null. This object must then be assigned before it can be used, exactly as when you declare a pointer.

`public` **`IloGoal`**`(IloGoalI * impl=0)`

This constructor creates a goal from its implementation object.

Methods

`public void end() const`

This member function ends the corresponding goal and returns the memory to the environment.

`public IloEnv getEnv() const`

This member function returns the environment to which the invoking goal belongs. A goal belongs to exactly one environment; different environments can not share the same goal.

`public IloGoalI * getImpl() const`

This member function returns a pointer to the implentation object of the invoking goal. This member function is useful when you need to be sure that you are using the same copy of the invoking goal in more than one situation.

`public const char * getName() const`

This member function returns a character string specifying the name of the invoking object (if there is one).

`public IloAny getObject() const`

This member function returns the object associated with the invoking object (if there is one). Normally, an associated object contains user data pertinent to the invoking object.

`public void setName(const char * name) const`

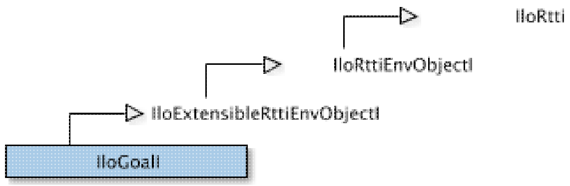
This member function assigns `name` to the invoking object.

`public void setObject(IloAny obj) const`

This member function associates `obj` with the invoking object. The member function `getObject` accesses this associated object afterward. Normally, `obj` contains user data pertinent to the invoking object.

Class IloGoalI

Definition file: ilsolver/ilosolverhandle.h
Include file: <ilsolver/ilosolver.h>



The class `IloGoal` represents goals in an IBM® ILOG® Concert Technology *model*. The class `IlcGoal` represents goals internally in a Solver search. A goal is a building block in a Solver search.

A goal is an object in Solver. Like other Solver entities, a goal is implemented by means of two classes: a handle class and an implementation class. In other words, an instance of the class `IloGoal` (a handle) contains a data member (the handle pointer) that points to an instance of the class `IloGoalI` (its implementation object).

See Also: IloGoal, ILOCPGOALWRAPPER0

Constructor and Destructor Summary	
`public`	`IloGoalI(IloEnvI *)`
`public`	`~IloGoalI()`

Method Summary	
`public virtual void`	`display(ostream &) const`
`public virtual IlcGoal`	`extract(const IloSolver solver) const`
`public virtual IloGoalI *`	`makeClone(IloEnvI * env) const`

Constructors and Destructors

`public` **`IloGoalI`**`(IloEnvI *)`

This constructor creates an instance of the class `IloGoalI`. This constructor should not be called directly as this class is an abstract class. This constructor is called automatically in the constructor of its subclasses.

`public` **`~IloGoalI`**`()`

This destructor is called automatically by the destructor of its subclasses. It frees memory used by the invoking object.

Methods

`public virtual void` **`display`**`(ostream &) const`

This member function prints the invoking goal on an output stream.

`public virtual IlcGoal` **`extract`**`(const IloSolver solver) const`

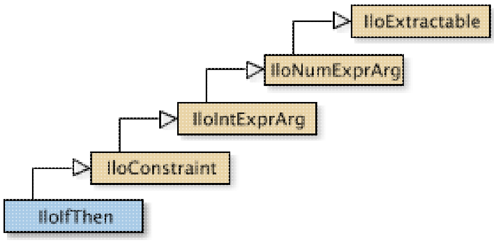
In general terms, in Concert Technology, the objects of a model must be extracted for an algorithm (an instance of one of the subclasses of `IloAlgorithm`, such as `IloSolver`). This member function returns the internal goal extracted for `solver` from the invoking goal of a model.

```
public virtual IloGoalI * makeClone(IloEnvI * env) const
```

This member function is called internally to duplicate the current goal

Class IloIfThen

Definition file: ilconcert/ilomodel.h



This class represents a condition constraint.
An instance of `IloIfThen` represents a condition constraint. Generally, a condition constraint is composed of an if part (the conditional statement or left side) and a then part (the consequence or right side).

In order for a constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloConstraint

Constructor Summary	
public	`IloIfThen()`
public	`IloIfThen(IloIfThenI * impl)`
public	`IloIfThen(const IloEnv env, const IloConstraint left, const IloConstraint right, const char * name=0)`

Method Summary	
`public IloIfThenI *`	`getImpl() const`

Inherited Methods from `IloConstraint`
`getImpl`

Inherited Methods from `IloIntExprArg`
`getImpl`

Inherited Methods from `IloNumExprArg`
`getImpl`

Inherited Methods from `IloExtractable`
`asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject`

Constructors

`public IloIfThen()`

This constructor creates an empty handle. You must initialize it before you use it.

`public IloIfThen(IloIfThenI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public IloIfThen(const IloEnv env, const IloConstraint left, const IloConstraint right, const char * name=0)`

This constructor creates a condition constraint in the environment specified by `env`. The argument `left` specifies the if-part of the condition. The argument `right` specifies the then-part of the condition. The string `name` specifies the name of the constraint; it is `0` (zero) by default. For the constraint to take effect, you must add it to a model and extract the model for an algorithm.

Methods

`public IloIfThenI * getImpl() const`

This member function returns a pointer to the implementation object of the invoking handle.

Class IloIIM

Definition file: ilsolver/iimextr.h

Management class for IIM components.
This class offers iterative improvement method (IIM) management facilities in much the same way as the `IloSolver` class does for pure constraint programming. This class is created from a solver and has the same life cycle as the solver.

See Also: IloNeighborIdentifier, IlcNeighborIdentifier

Constructor Summary	
`public`	`IloIIM(IloSolver solver)` This constructor builds an IIM management class from a solver.

Method Summary	
`public IloSolution`	`getAtDelta(const IloNeighborIdentifier nid) const` This member function returns the solution delta of the neighbor accepted in the local search using neighbor identifier `nid`.
`public IloInt`	`getAtIndex(const IloNeighborIdentifier nid) const` This member function returns the index of the neighbor accepted in the local search using neighbor identifier `nid`.
`public IlcNeighborIdentifier`	`getNeighborIdentifier(const IloNeighborIdentifier nid) const` This member function returns the neighbor identifier extracted from `nid`.
`public IloSolver`	`getSolver() const` Delivers the solver passed at construction time.

Constructors

`public` **`IloIIM`**`(IloSolver solver)`

This constructor builds an IIM management class from a solver.

This constructor builds an IIM management class from the solver `solver`. The life cycle of the constructed class will be equivalent to that of the solver. When the solver is destroyed, the IIM management object must no longer be used. If more than one instance of `IloIIM` is created on the same solver, then these instances will share the same implementation pointer.

Methods

`public IloSolution` **`getAtDelta`**`(const IloNeighborIdentifier nid) const`

This member function returns the solution delta of the neighbor accepted in the local search using neighbor identifier `nid`.

This member function returns the solution delta of the neighbor accepted in the local search using neighbor identifier `nid`. The solution delta is specified by the neighborhood and is the set of extractables which change value together with their new values. This is equivalent to `return getNeighborIdentifier(nid).getAtDelta();` The returned delta *does not* belong to the user, but the neighborhood identifier itself: `end()` must not be called on it.

`public IloInt **getAtIndex**(const IloNeighborIdentifier nid) const`

This member function returns the index of the neighbor accepted in the local search using neighbor identifier `nid`.

This member function returns the index of the neighbor accepted in the local search using neighbor identifier `nid`. This is equivalent to `return getNeighborIdentifier(nid).getAtIndex();`

`public IlcNeighborIdentifier **getNeighborIdentifier**(const IloNeighborIdentifier nid) const`

This member function returns the neighbor identifier extracted from `nid`.

This member function returns the neighborhood identifier used inside search which is extracted from the neighborhood identifier `nid` used for search specification.

`public IloSolver **getSolver**() const`

Delivers the solver passed at construction time.

This member functions returns the solver passed at construction time.

Class IloCsvReader::IloIncorrectCsvReaderUseException

Definition file: ilconcert/ilocsvreader.h

Exception thrown for call to inappropriate csv reader.
This exception is thrown in the following member functions if you call them from a reader built as a multitable csv reader.

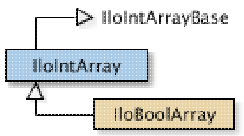
- IloCsvReader::getLineByNumber
- IloCsvReader::getLineByKey
- IloCsvReader::getNumberOfItems
- IloCsvReader::getNumberOfColumns
- IloCsvReader::getNumberOfKeys
- IloCsvReader::getReaderForUniqueTableFile
- IloCsvReader::getTable
- IloCsvReader::isHeadingExists
- IloCsvReader::printKeys

This exception is throw in the following member functions if you call them from a reader built as a unique table csv reader.

- IloCsvReader::getCsvFormat
- IloCsvReader::getFileVersion
- IloCsvReader::getTableByName
- IloCsvReader::getTableByNumber
- IloCsvReader::getRequiredBy

Class IloIntArray

Definition file: ilconcert/iloenv.h



The array class of the basic integer class.
`IloIntArray` is the array class of the basic integer class for a model. It is a handle class. The implementation class for `IloIntArray` is the undocumented class `IloIntArrayI`.

Instances of `IloIntArray` are extensible. (They differ from instances of `IlcIntArray` in this respect.) References to an array change whenever an element is added to or removed from the array.

For each basic type, Concert Technology defines a corresponding array class. That array class is a handle class. In other words, an object of that class contains a pointer to another object allocated in a Concert Technology environment associated with a model. Exploiting handles in this way greatly simplifies the programming interface since the handle can then be an automatic object: as a developer using handles, you do not have to worry about memory allocation.

As handles, these objects should be passed by value, and they should be created as automatic objects, where "automatic" has the usual C++ meaning.

Member functions of a handle class correspond to member functions of the same name in the implementation class.

Assert and NDEBUG

Most member functions of the class `IloIntArray` are inline functions that contain an `assert` statement. This statement checks that the handle pointer is not null. These statements can be suppressed by the macro `NDEBUG`. This option usually reduces execution time. The price you pay for this choice is that attempts to access through null pointers are not trapped and usually result in memory faults.

`IloIntArray` inherits additional methods from the template `IloArray`:

- `IloArray::add`
- `IloArray::add`
- `IloArray::clear`
- `IloArray::getEnv`
- `IloArray::getSize`
- `IloArray::remove`
- `IloArray::operator[]`
- `IloArray::operator[]`

See Also: IloInt

Constructor Summary	
`public`	`IloIntArray(IloArrayI * i=0)`
`public`	`IloIntArray(const IloEnv env, IloInt n=0)`
`public`	`IloIntArray(const IloEnv env, IloInt n, IloInt v0, IloInt v1...)`

Method Summary	
`public IloBool`	`contains(IloIntArray ax) const`

public IloBool	contains(IloInt value) const
public void	discard(IloIntArray ax)
public void	discard(IloInt value)
public IloIntExprArg	operator[](IloIntExprArg intExp) const
public IloInt &	operator[](IloInt i)
public const IloInt &	operator[](IloInt i) const
public IloNumArray	toNumArray() const

Constructors

`public` **`IloIntArray`**`(IloArrayI * i=0)`

This constructor creates an array of integers from an implementation object.

`public` **`IloIntArray`**`(const IloEnv env, IloInt n=0)`

This constructor creates an array of `n` integers for use in a model in the environment specified by `env`. By default, its elements are empty handles.

`public` **`IloIntArray`**`(const IloEnv env, IloInt n, IloInt v0, IloInt v1...)`

This constructor creates an array of `n` integers; the elements of the new array take the corresponding values: `v0, v1, ..., v(n-1).`

Methods

`public IloBool` **`contains`**`(IloIntArray ax) const`

This member function checks whether all the values of `ax` are contained or not.

`public IloBool` **`contains`**`(IloInt value) const`

This member function checks whether the value is contained or not.

`public void` **`discard`**`(IloIntArray ax)`

This member function removes elements from the invoking array. It removes the array `ax`.

`public void` **`discard`**`(IloInt value)`

This member function removes elements from the invoking array. It removes the element.

`public IloIntExprArg` **`operator[]`**`(IloIntExprArg intExp) const`

This subscripting operator returns an expression node for use in a constraint or expression. For clarity, let's call `A` the invoking array. When `intExp` is bound to the value `i`, then the domain of the expression is the domain of `A[i]`. More generally, the domain of the expression is the union of the domains of the expressions `A[i]` where the `i` are in the domain of `intExp`.

This operator is also known as an element expression.

`public IloInt & `**`operator[]`**`(IloInt i)`

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`.

`public const IloInt & `**`operator[]`**`(IloInt i) const`

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`. On `const` arrays, Concert Technology uses the `const` operator:

```
IloArray operator[] (IloInt i) const;
```

`public IloNumArray `**`toNumArray`**`() const`

This constructor creates an array of integers from an array of numeric values.

Class IloIntBinaryPredicate

Definition file: ilconcert/ilotupleset.h

For constraint programming: binary predicates operating on arbitrary objects in a model.
This class makes it possible for you to define binary predicates operating on arbitrary objects in a model. A predicate is an object with a member function (such as `IloIntBinaryPredicate::isTrue`) that checks whether or not a property is satisfied by an ordered set of (pointers to) objects.

Defining a New Class of Predicates

Predicates, like other Concert Technology objects, depend on two classes: a handle class, `IloIntBinaryPredicate`, and an implementation class, such as `IloIntBinaryPredicateI`, where an object of the handle class contains a data member (the handle pointer) that points to an object (its implementation object) of an instance of `IloIntBinaryPredicateI` allocated in a Concert Technology environment. As a Concert Technology user, you will be working primarily with handles.

If you define a new class of predicates yourself, you must define its implementation class together with the corresponding virtual member function `isTrue`, as well as a member function that returns an instance of the handle class `IloIntBinaryPredicate`.

Arity

As a developer, you can use predicates in Concert Technology applications to define your own constraints that have not already been predefined in Concert Technology. In that case, the *arity* of the predicate (that is, the number of constrained variables involved in the predicate, and thus the size of the array that the member function `IloIntBinaryPredicate::isTrue` must check) must be two.

See Also these classes in the *IBM ILOG CP Optimizer Reference Manual*: `IloAllowedAssignments`, `IloForbiddenAssignments`.

See Also these classes in the *IBM ILOG Solver Reference Manual*: `IloTableConstraint`.

Constructor and Destructor Summary	
`public`	`IloIntBinaryPredicate()`
`public`	`IloIntBinaryPredicate(IloIntBinaryPredicateI * impl)`

Method Summary	
`public IloIntBinaryPredicateI *`	`getImpl() const`
`public IloBool`	`isTrue(const IloInt val1, const IloInt val2)`
`public void`	`operator=(const IloIntBinaryPredicate & h)`

Constructors and Destructors

`public` **`IloIntBinaryPredicate`**`()`

This constructor creates an empty binary predicate. In other words, the predicate is an empty handle with a null handle pointer. You must assign the elements of the predicate before you attempt to access it, just as you would any other pointer. Int attempt to access it before this assignment will throw an exception (an instance of `IloSolver::SolverErrorException`).

```
public IloIntBinaryPredicate(IloIntBinaryPredicateI * impl)
```

This constructor creates a handle object (an instance of the class `IloIntBinaryPredicate`) from a pointer to an implementation object (an instance of the implementation class `IlcIntPredicateI`, documented in the *IBM ILOG CP Optimizer Reference Manual* and the *IBM ILOG Solver Reference Manual*).

Methods

```
public IloIntBinaryPredicateI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloBool isTrue(const IloInt val1, const IloInt val2)
```

This member function returns `IloTrue` if the values `val1` and `val2` make the invoking binary predicate valid. It returns `IloFalse` otherwise.

```
public void operator=(const IloIntBinaryPredicate & h)
```

This assignment operator copies `h` into the invoking predicate by assigning an address to the handle pointer of the invoking object. That address is the location of the implementation object of the argument `h`. After execution of this operator, both the invoking predicate and `h` point to the same implementation object.

Class IloIntervalList

Definition file: ilconcert/ilointervals.h

Represents a list of nonoverlapping intervals.
An instance of the class `IloIntervalList` represents a list of nonoverlapping intervals. Each interval `[timeMin, timeMax)` from the list is associated with a numeric type.

Note that if *n* is the number of intervals in the list, the random access to a given interval (see the member functions `IloIntervalList::addInterval`, `IloIntervalList::contains`, and `IloIntervalList::removeInterval`) has a worst-case complexity in *O(log(n))*.

Furthermore, when two consecutive intervals of the list have the same types, these intervals are merged so that the list is always represented with the minimal number of intervals.

See Also: IloIntervalListCursor, IloUnion, IloDifference

Constructor Summary	
`public`	`IloIntervalList(const IloEnv env, IloNum min=-IloInfinity, IloNum max=+IloInfinity, const char * name=0)`
`public`	`IloIntervalList(const IloEnv env, const IloNumArray times, const IloNumArray types, const char * name=0)`

Method Summary	
`public void`	`addInterval(IloNum start, IloNum end, IloNum type=0L) const`
`public void`	`addPeriodicInterval(IloNum start, IloNum duration, IloNum period, IloNum end, IloNum type=0L) const`
`public IloBool`	`contains(const IloIntervalList intervals) const`
`public IloIntervalList`	`copy() const`
`public void`	`dilate(IloNum k) const`
`public void`	`empty() const`
`public IloNum`	`getDefinitionIntervalMax() const`
`public IloNum`	`getDefinitionIntervalMin() const`
`public IloBool`	`isEmpty() const`
`public IloBool`	`isKeptOpen() const`
`public void`	`keepOpen(IloBool val=IloTrue) const`
`public void`	`removeInterval(IloNum start, IloNum end) const`
`public void`	`removeIntervalOnDuration(IloNum start, IloNum duration) const`
`public void`	`removePeriodicInterval(IloNum start, IloNum duration, IloNum period, IloNum end) const`
`public void`	`setDifference(const IloIntervalList intervals) const`
`public void`	`setPeriodic(const IloIntervalList intervals, IloNum x0, IloNum n=IloInfinity) const`
`public void`	`setUnion(const IloIntervalList intervals) const`

public void	shift(IloNum dx) const

Constructors

`public IloIntervalList(const IloEnv env, IloNum min=-IloInfinity, IloNum max=+IloInfinity, const char * name=0)`

This constructor creates a new instance of `IloIntervalList` and adds it to the set of interval lists managed in the given environment. The arguments `min` and `max` respectively represent the origin and the horizon of the interval list. The new interval list does not contain any intervals.

`public IloIntervalList(const IloEnv env, const IloNumArray times, const IloNumArray types, const char * name=0)`

This constructor creates an interval list whose intervals are defined by the two arrays `times` and `types`. More precisely, if `n` is the size of array `times`, then the size of array `types` must be `n-1` and the following contiguous intervals are created on the interval list: `[times[i],times[i+1])` with type `types[i]` for all `i` in `[0, n-1]`.

Methods

`public void addInterval(IloNum start, IloNum end, IloNum type=0L) const`

This member function adds an interval of type `type` to the invoking interval list. The start time and end time of that newly added interval are set to `start` and `end`. By default, the type of the interval is `0`. Adding a new interval that overlaps with an already existing interval of a different type will override the existing type on the intersection.

`public void addPeriodicInterval(IloNum start, IloNum duration, IloNum period, IloNum end, IloNum type=0L) const`

This member function adds a set of intervals to the invoking interval list. For every `i >= 0` such that `start + i * period < end`, an interval of `[start + i * period, start + duration + i * period)` is added. By default, the type of these intervals is `0`. Adding a new interval that overlaps with an already existing interval of a different type will override the existing type on the intersection.

`public IloBool contains(const IloIntervalList intervals) const`

This member function returns `IloTrue` if and only if each interval of `intervals` is included in an interval of the invoking interval list, regardless of interval type.

`public IloIntervalList copy() const`

This member function creates and returns a new interval list that is a copy of the invoking interval list.

`public void dilate(IloNum k) const`

This member function multiplies by `k` the scale of times for the invoking interval list. `k` must be a positive number.

`public void **empty**() const`

This member function removes all the intervals from the invoking interval list.

`public IloNum **getDefinitionIntervalMax**() const`

This member function returns the right most point (horizon) of the definition interval of the invoking interval list.

`public IloNum **getDefinitionIntervalMin**() const`

This member function returns the left most point (origin) of the definition interval of the invoking interval list.

`public IloBool **isEmpty**() const`

This member function returns `IloTrue` if and only if the invoking interval list is empty.

`public IloBool **isKeptOpen**() const`

This member function returns `IloTrue` if the interval list must be kept open. Otherwise, it returns `IloFalse`.

`public void **keepOpen**(IloBool val=IloTrue) const`

If the argument `val` is equal to `IloTrue`, this member function states that the invoking interval list must be kept open during the search for a solution to the problem. It means that additional intervals may be added during the search. Otherwise, if the argument `val` is equal to `IloFalse`, it states that all the intervals of the invoking interval list will be defined in the model before starting to solve the problem. By default, it is supposed that all the intervals of the invoking interval list are defined in the model before starting to solve the problem.

`public void **removeInterval**(IloNum start, IloNum end) const`

This member function removes all intervals on the invoking interval list between `start` and `end`. If start is placed inside an interval `[start1, end1)`, that is, `start1 < start < end1`, this results in an interval `[start1, start)`. If `end` is placed inside an interval `[start2, end2)` this results in an interval `[end, end2)`.

`public void **removeIntervalOnDuration**(IloNum start, IloNum duration) const`

This member function removes all intervals on the invoking resource between `start` and `start+duration`.

`public void **removePeriodicInterval**(IloNum start, IloNum duration, IloNum period, IloNum end) const`

This member function removes intervals from the invoking interval list. More precisely, for every `i >= 0` such that `start + i * period < end`, this function removes all intervals between `start + i * period` and `start + duration + i * period`.

`public void` **`setDifference`**`(const IloIntervalList intervals) const`

This member function removes from the invoking interval list all the `intervals` contained in the interval list `intervals`. The definition interval of the invoking interval list is not changed.

`public void` **`setPeriodic`**`(const IloIntervalList intervals, IloNum x0, IloNum n=IloInfinity) const`

This member function initializes the invoking interval list as an interval list that repeats the interval list intervals `n` times after `x0`.

`public void` **`setUnion`**`(const IloIntervalList intervals) const`

This member function sets the invoking interval list to be the union between the current interval list and the interval list `intervals`. An instance of `IloException` is thrown if two intervals with different types overlap. The definition interval of the invoking interval list is set to the union between the current definition interval and the definition interval of `intervals`.

`public void` **`shift`**`(IloNum dx) const`

This member function shifts the intervals of the invoking interval list from `dx` to the right if `dx > 0` or from `-dx` to the left if `dx < 0`. It has no effect if `dx = 0`.

Class IloIntervalListCursor

Definition file: ilconcert/ilointervals.h

Inspects the intervals of an interval list.
An instance of the class `IloIntervalListCursor` allows you to inspect the intervals of an interval list, that is, an instance of `IloIntervalList`. Cursors are intended to iterate forward or backward over the intervals of an interval list.

> **Note**
>
> The structure of the interval list cannot be changed while a cursor is being used to inspect it. Therefore, functions that change the structure of the interval list, such as IloIntervalList::addInterval, should not be called while the cursor is being used.

See Also: IloIntervalList

Constructor and Destructor Summary	
`public`	`IloIntervalListCursor(const IloIntervalList)`
`public`	`IloIntervalListCursor(const IloIntervalList, IloNum x)`
`public`	`IloIntervalListCursor(const IloIntervalListCursor &)`

Method Summary	
`public IloNum`	`getEnd() const`
`public IloNum`	`getStart() const`
`public IloNum`	`getType() const`
`public IloBool`	`ok() const`
`public void`	`operator++()`
`public void`	`operator--()`
`public void`	`operator=(const IloIntervalListCursor &)`
`public void`	`seek(IloNum)`

Constructors and Destructors

`public` **`IloIntervalListCursor`**`(const IloIntervalList)`

This constructor creates a cursor to inspect the interval list argument. This cursor lets you iterate forward or backward over the intervals of the interval list. The cursor initially specifies the first interval of the interval list.

`public` **`IloIntervalListCursor`**`(const IloIntervalList, IloNum x)`

This constructor creates a cursor to inspect the interval list `intervals`. This cursor lets you iterate forward or backward over the interval list. The cursor initially specifies the interval of the interval list that contains `x`.

Note that if *n* is the number of intervals of the interval list given as argument, the worst-case complexity of this constructor is *O(log(n))*.

`public IloIntervalListCursor(const IloIntervalListCursor &)`

This constructor creates a new cursor that is a copy of the argument. The new cursor initially specifies the same interval and the same interval list as the argument `cursor`.

Methods

`public IloNum getEnd() const`

This member function returns the end point of the interval currently specified by the cursor.

`public IloNum getStart() const`

This member function returns the start point of the interval currently specified by the cursor.

`public IloNum getType() const`

This member function returns the type of the interval currently specified by the cursor.

`public IloBool ok() const`

This member function returns `IloFalse` if the cursor does not currently specify an interval included in the interval list. Otherwise, it returns `IloTrue`.

`public void operator++()`

This operator moves the cursor to the interval adjacent to the current interval (forward move).

`public void operator--()`

This operator moves the cursor to the interval adjacent to the current interval (backward move).

`public void operator=(const IloIntervalListCursor &)`

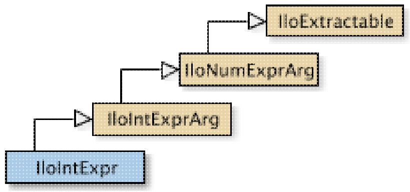
This operator assigns an address to the handle pointer of the invoking instance of `IloIntervalListCursor`. That address is the location of the implementation object of the argument `cursor`. After the execution of this operator, the invoking object and `cursor` both point to the same implementation object.

`public void seek(IloNum)`

This member function sets the cursor to specify the first interval of the interval list whose end is strictly greater than the argument. Note that if *n* is the number of intervals of the interval list traversed by the invoking iterator, the worst-case complexity of this member function is *O(log(n))*. An instance of `IloException` is thrown if the argument does not belong to the interval of definition of the invoking interval list.

Class IloIntExpr

Definition file: ilconcert/iloexpression.h



The class of integer expressions in Concert Technology.
Integer expressions in Concert Technology are represented using objects of type `IloIntExpr`.

Constructor Summary	
public	`IloIntExpr()`
public	`IloIntExpr(IloIntExprI * impl)`
public	`IloIntExpr(const IloIntExprArg arg)`
public	`IloIntExpr(const IloIntLinExprTerm term)`
public	`IloIntExpr(const IloEnv env, IloInt constant=0)`

Method Summary	
public IloIntExprI *	`getImpl() const`
public IloIntExpr &	`operator*=(IloInt val)`
public IloIntExpr &	`operator+=(const IloIntExprArg expr)`
public IloIntExpr &	`operator+=(IloInt val)`
public IloIntExpr &	`operator-=(const IloIntExprArg expr)`
public IloIntExpr &	`operator-=(IloInt val)`

Inherited Methods from `IloIntExprArg`
getImpl

Inherited Methods from `IloNumExprArg`
getImpl

Inherited Methods from `IloExtractable`
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Constructors

public **IloIntExpr**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IloIntExpr**(IloIntExprI * impl)

This constructor creates a handle object from a pointer to an implementation object.

public **IloIntExpr**(const IloIntExprArg arg)

This constructor creates an integer expression using the undocumented class IloIntExprArg.

public **IloIntExpr**(const IloIntLinExprTerm term)

This constructor creates an integer expression with linear terms using the undocumented class IloIntLinExprTerm.

public **IloIntExpr**(const IloEnv env, IloInt constant=0)

This constructor creates a constant integer expression with the value constant that the user can modify subsequently with the operators +=, -=, = in the environment env.

Methods

public IloIntExprI * **getImpl**() const

This member function returns a pointer to the implementation object of the invoking handle.

public IloIntExpr & **operator*=**(IloInt val)

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than x = x * ....

public IloIntExpr & **operator+=**(const IloIntExprArg expr)

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than x = x + ...

public IloIntExpr & **operator+=**(IloInt val)

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than x = x + ...

public IloIntExpr & **operator-=**(const IloIntExprArg expr)

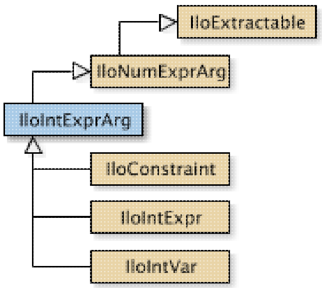
This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than x = x - ....

public IloIntExpr & **operator-=**(IloInt val)

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than x = x - ....

Class IloIntExprArg

Definition file: ilconcert/iloexpression.h



A class used internally in Concert Technology.
Concert Technology uses instances of these classes internally as temporary objects when it is parsing a C++ expression in order to build an instance of `IloIntExpr`. As a Concert Technology user, you will not need this class yourself; in fact, you should not use them directly. They are documented here because the return value of certain functions, such as `IloSum` or `IloScalProd`, can be an instance of this class.

Constructor Summary	
public	IloIntExprArg()
public	IloIntExprArg(IloIntExprI * impl)

Method Summary	
public IloIntExprI *	getImpl() const

Inherited Methods from `IloNumExprArg`
getImpl

Inherited Methods from `IloExtractable`
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Constructors

`public` **`IloIntExprArg`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IloIntExprArg`**`(IloIntExprI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

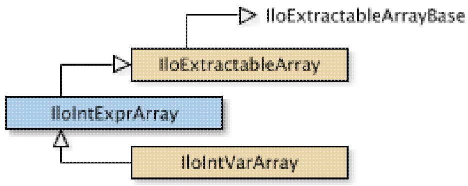
Methods

`public IloIntExprI *` **`getImpl`**`() const`

This member function returns a pointer to the implementation object of the invoking handle.

Class IloIntExprArray

Definition file: ilconcert/iloexpression.h



The array class of the integer expressions class.
For each basic type, Concert Technology defines a corresponding array class. `IloIntExprArray` is the array class of the integer expressions class (`IloIntExpr`) for a model.

Instances of `IloIntExprArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Constructor Summary	
public	IloIntExprArray(IloDefaultArrayI * i=0)
public	IloIntExprArray(const IloEnv env, IloInt n=0)

Method Summary	
public void	add(IloInt more, const IloIntExpr x)
public void	add(const IloIntExpr x)
public void	add(const IloIntExprArray array)
public void	endElements()
public IloIntExprArg	operator[](IloIntExprArg anIntegerExpr) const
public IloIntExpr	operator[](IloInt i) const
public IloIntExpr &	operator[](IloInt i)

Inherited Methods from `IloExtractableArray`
add, add, add, endElements, setNames

Constructors

`public` **`IloIntExprArray`**`(IloDefaultArrayI * i=0)`

This constructor creates an empty array of elements. You cannot create instances of the undocumented class `IloDefaultArrayI`. As an argument in this default constructor, it allows you to pass 0 (zero) as a value to an optional argument in functions and member functions that accept an array as an argument.

`public` **`IloIntExprArray`**`(const IloEnv env, IloInt n=0)`

This constructor creates an array of n elements. Initially, the n elements are empty handles.

Methods

`public void add(IloInt more, const IloIntExpr x)`

This member function appends `x` to the invoking array multiple times. The argument `more` specifies how many times.

`public void add(const IloIntExpr x)`

This member function appends `x` to the invoking array.

`public void add(const IloIntExprArray array)`

This member function appends the elements in `array` to the invoking array.

`public void endElements()`

This member function calls `IloExtractable::end` for each of the elements in the invoking array. This deletes all the extractables identified by the array, leaving the handles in the array intact. This member function is the recommended way to delete the elements of an array.

`public IloIntExprArg operator[](IloIntExprArg anIntegerExpr) const`

This subscripting operator returns an expression argument for use in a constraint or expression. For clarity, let's call `A` the invoking array. When `anIntegerExpr` is bound to the value `i`, the domain of the expression is the domain of `A[i]`. More generally, the domain of the expression is the union of the domains of the expressions `A[i]` where the `i` are in the domain of `anIntegerExpr`.

This operator is also known as an element expression.

`public IloIntExpr operator[](IloInt i) const`

This operator returns a reference to the extractable object located in the invoking array at the position specified by the index `i`. On `const` arrays, Concert Technology uses the `const` operator:

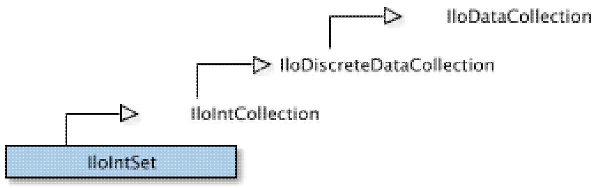
```
IloIntExpr operator[] (IloInt i) const;
```

`public IloIntExpr & operator[](IloInt i)`

This operator returns a reference to the extractable object located in the invoking array at the position specified by the index `i`.

Class IloIntSet

Definition file: ilconcert/iloset.h



An instance of this class offers a convenient way to represent a set of integer values.
An instance of this class offers a convenient way to represent a set of integer values as a constrained variable in Concert Technology.

An instance of this class represents a set of enumerated values. The same enumerated value will not appear more than once in a set. The elements of a set are not ordered. The class `IloIntSet::Iterator` offers you a way to traverse the elements of such a set.

If you are considering modeling issues where you want to represent repeated elements or where you want to exploit an indexed order among the elements, then you might want to look at the class `IloAnyArray` instead of this class for sets.

See Also: IloExtractable, IloModel, IloIntSetVarArray

Constructor Summary	
public	IloIntSet(const IloEnv env, const IloIntArray array, IloBool withIndex=IloFalse)
public	IloIntSet(const IloEnv env, const IloNumArray array, IloBool withIndex=IloFalse)
public	IloIntSet(const IloEnv env, IloBool withIndex=IloFalse)
public	IloIntSet(IloIntSetI * impl=0)

Method Summary	
public void	add(IloIntSet set)
public void	add(IloInt elt)
public IloBool	contains(IloIntSet set) const
public IloBool	contains(IloInt elt) const
public void	empty()
public IloInt	getFirst() const
public IloIntSetI *	getImpl() const
public IloInt	getLast() const
public IloInt	getNext(IloInt value, IloInt offset=1) const
public IloInt	getNextC(IloInt value, IloInt offset=1) const
public IloInt	getPrevious(IloInt value, IloInt offset=1) const
public IloInt	getPreviousC(IloInt value, IloInt offset=1) const
public IloInt	getSize() const
public IloBool	intersects(IloIntSet set) const
public void	remove(IloIntSet set)

public void	remove(IloInt elt)
public void	setIntersection(IloIntSet set)
public void	setIntersection(IloInt elt)

Inner Class	
IloIntSet::Iterator	This class is an iterator that traverses the elements of a finite set of numeric values.

Constructors

`public IloIntSet(const IloEnv env, const IloIntArray array, IloBool withIndex=IloFalse)`

This constructor creates a set of integer values in the environment `env` from the elements in `array`. The optional flag `withIndex` corresponds to the activation or not of internal Hash Tables to improve speed of `add`/`getIndex` methods.

`public IloIntSet(const IloEnv env, const IloNumArray array, IloBool withIndex=IloFalse)`

This constructor creates a set of numeric values in the environment `env` from the elements in `array`. The optional flag `withIndex` corresponds to the activation or not of internal Hash Tables to improve speed of `add`/`getIndex` methods.

`public IloIntSet(const IloEnv env, IloBool withIndex=IloFalse)`

This constructor creates an empty set (no elements) in the environment `env`. You must use the member function `IloIntSet::add` to fill this set with elements. The optional flag `withIndex` corresponds to the activation or not of internal Hash Tables to improve speed of `add`/`getIndex` methods.

`public IloIntSet(IloIntSetI * impl=0)`

This constructor creates a handle to a set of integer values from its implementation object.

Methods

`public void add(IloIntSet set)`

This member function adds `set` to the invoking set. Here, "adds" means that the invoking set becomes the union of its former elements and the elements of `set`.

To calculate the arithmetic sum of values in an array, use the function `IloSum`.

`public void add(IloInt elt)`

This member function adds `elt` to the invoking set. Here, "adds" means that the invoking set becomes the union of its former elements and the new `elt`.

`public IloBool contains(IloIntSet set) const`

This member function returns a Boolean value (zero or one) that specifies whether `set` contains the invoking set. The value one specifies that the invoking set contains all the elements of set, and that the intersection of the invoking set with `set` is precisely `set`. The value zero specifies that the intersection of the invoking set and `set` is not precisely `set`.

`public IloBool contains(IloInt elt) const`

This member function returns a Boolean value (zero or one) that specifies whether `elt` is an element of the invoking set. The value one specifies that the invoking set contains `elt`; the value zero specifies that the invoking set does not contain `elt`.

`public void empty()`

This member function removes the elements from the invoking set. In other words, the invoking set becomes the empty set.

`public IloInt getFirst() const`

Returns the first item of the collection.

`public IloIntSetI * getImpl() const`

This member function returns a pointer to the implementation object of the invoking set.

`public IloInt getLast() const`

Returns the last item of the collection.

`public IloInt getNext(IloInt value, IloInt offset=1) const`

This method returns the value next to the given argument in the set.

If the given value does not exist, it throws an exception

If no value follows (that is, you are at the end of the set), it throws an exception.

See also getNextC, getPreviousC for circular search.

```
S = {1,2,3,4}
S.next(2,1) will return 3
```

`public IloInt getNextC(IloInt value, IloInt offset=1) const`

This method returns the value next to the given argument in the set.

If the given value does not exist, it throws an exception.

If no value follows (that is, you are at the end of the set), it will give you the first value (circular search).

See also getNext, getPrevious.

See Also: IloIntSet::getNext

```
public IloInt getPrevious(IloInt value, IloInt offset=1) const
```

This method returns the value previous to the given argument in the set.

If the given value does not exist, it throws an exception

If no value is previous (that is, you are at the beginning of the set), it throws an exception.

See also getNextC, getPreviousC for circular search.

See Also: IloIntSet::getNext

```
public IloInt getPreviousC(IloInt value, IloInt offset=1) const
```

This method returns the value previous to the given argument in the set.

If the given value does not exist, it throws an exception.

If no value is previous (that is, you are at the beginning of the set), it will give you the last value (circular search).

See also getNext, getPrevious.

See Also: IloIntSet::getNext

```
public IloInt getSize() const
```

This member function returns an integer specifying the size of the invoking set (that is, how many elements it contains).

```
public IloBool intersects(IloIntSet set) const
```

This member function returns a Boolean value (zero or one) that specifies whether `set` intersects the invoking set. The value one specifies that the intersection of `set` and the invoking set is not empty (at least one element in common); the value zero specifies that the intersection of `set` and the invoking set is empty (no elements in common).

```
public void remove(IloIntSet set)
```

This member function removes all the elements of `set` from the invoking set.

```
public void remove(IloInt elt)
```

This member function removes `elt` from the invoking set.

```
public void setIntersection(IloIntSet set)
```

This member function changes the invoking set so that it includes only the elements of `set`. In other words, the invoking set becomes the intersection of its former elements with the elements of `set`.

```
public void setIntersection(IloInt elt)
```

This member function changes the invoking set so that it includes only the element specified by `elt`. In other words, the invoking set becomes the intersection of its former elements with `elt`.

Class IloIntSetValueSelector

Definition file: ilsolver/ilosolverset.h
Include file: <ilsolver/ilosolver.h>

Solver lets you create value selectors to control the order in which the values in the domain of a set of constrained integer variables are tried during the search for a solution.

The class `IloIntSetValueSelector` represents value selectors in an IBM® ILOG® Concert Technology *model*. The class `IlcIntSetSelect` represents value selectors internally in a Solver search.

This class is the handle class of the modeling object that wraps the search object. When search starts, `IloIntSetValueSelectorI` is extracted into an instance of `IlcIntSetSelectI`.

See Also: IlcIntSetSelect, IloIntSetValueSelectorI

Constructor Summary	
`public`	`IloIntSetValueSelector()`
`public`	`IloIntSetValueSelector(IloIntSetValueSelectorI * impl)`

Method Summary	
`public IloIntSetValueSelectorI *`	`getImpl() const`
`public void`	`operator=(const IloIntSetValueSelector & h)`

Constructors

`public` **`IloIntSetValueSelector`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IloIntSetValueSelector`**`(IloIntSetValueSelectorI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

Methods

`public IloIntSetValueSelectorI *` **`getImpl`**`() const`

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

`public void` **`operator=`**`(const IloIntSetValueSelector & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IloIntSetValueSelectorI

Definition file: ilsolver/ilosolverset.h
Include file: <ilsolver/ilosolver.h>

This class is the implementation class for `IloIntSetValueSelector`.

The class `IloIntSetValueSelectorI` is the implementation class for value selectors in an IBM® ILOG® Concert Technology *model*. The class `IlcIntSetSelectI` is the implementation class for value selectors internally in a Solver search.

This class is the modeling object that wraps the search object. When search starts, `IloIntSetValueSelectorI` is extracted into an instance of `IlcIntSetSelectI`.

To define new selection criteria, you define both a subclass of `IloIntSetValueSelectorI` and a subclass of `IlcIntSetSelectI`.

See Also: IlcIntSetSelect, IloIntSetValueSelector

Constructor and Destructor Summary	
`public`	`IloIntSetValueSelectorI(IloEnvI *)`
`public`	`~IloIntSetValueSelectorI()`

Method Summary	
`public virtual void`	`display(ostream &) const`
`public virtual IlcIntSetSelect`	`extract(const IloSolver solver) const`
`public IloEnvI *`	`getEnv() const`
`public virtual IloIntSetValueSelectorI *`	`makeClone(IloEnvI * env) const`

Constructors and Destructors

`public` **`IloIntSetValueSelectorI`**`(IloEnvI *)`

This constructor creates an instance of the class `IloIntSetValueSelectorI`. This constructor should not be called directly as this class is an abstract class. This constructor is called automatically in the constructor of its subclasses.

`public` **`~IloIntSetValueSelectorI`**`()`

This destructor is called automatically by the destructor of its subclasses. It frees memory used by the invoking object.

Methods

`public virtual void` **`display`**`(ostream &) const`

This member function prints the invoking value selector on an output stream.

`public virtual IlcIntSetSelect **extract**(const IloSolver solver) const`

In general terms, in Concert Technology, the objects of a model must be extracted for an algorithm (an instance of one of the subclasses of `IloAlgorithm`, such as `IloSolver`). This member function returns the internal value selector extracted for `solver` from the invoking value selector of a model.

`public IloEnvI * **getEnv**() const`

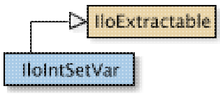
This member function returns the environment to which the invoking value selector belongs. A value selector belongs to exactly one environment; different environments cannot share the same value selector.

`public virtual IloIntSetValueSelectorI * **makeClone**(IloEnvI * env) const`

This member function is called internally to duplicate the current value selector.

Class IloIntSetVar

Definition file: ilconcert/iloset.h



The class `IloIntSetVar` represents a set of integer values.
An instance of this class represents a set of integer values. The same integer value will not appear more than once in a set. The elements of a set are not ordered.

A constrained variable representing a set of integer values (that is, an instance of `IloIntSetVar`) is defined in terms of two other sets: its required elements and its possible elements. Its required elements are those that must be in the set. Its possible elements are those that may be in the set. This class offers member functions for accessing the required and possible elements of a set of integer values.

The function `IloCard` offers you a way to constrain the number of elements in a set variable. That is, `IloCard` constrains the cardinality of a set variable.

Constructor Summary	
public	`IloIntSetVar()`
public	`IloIntSetVar(IloIntSetVarI * impl)`
public	`IloIntSetVar(const IloEnv env, const IloIntArray array, const char * name=0)`
public	`IloIntSetVar(const IloEnv env, const IloIntArray possible, const IloIntArray required, const char * name=0)`
public	`IloIntSetVar(const IloEnv env, const IloNumArray array, const char * name=0)`
public	`IloIntSetVar(const IloEnv env, const IloNumArray possible, const IloNumArray required, const char * name=0)`
public	`IloIntSetVar(const IloIntCollection possible, const char * name=0)`
public	`IloIntSetVar(const IloIntCollection possible, const IloIntCollection required, const char * name=0)`
public	`IloIntSetVar(const IloNumCollection possible, const char * name=0)`
public	`IloIntSetVar(const IloNumCollection possible, const IloNumCollection required, const char * name=0)`

Method Summary	
public void	`addPossible(IloInt elt) const`
public void	`addRequired(IloInt elt) const`
public IloIntSetVarI *	`getImpl() const`
public void	`getPossibleSet(IloIntSet set) const`
public IloIntSet	`getPossibleSet() const`
public void	`getRequiredSet(IloIntSet set) const`
public IloIntSet	`getRequiredSet() const`
public void	`removePossible(IloInt elt) const`
public void	`removeRequired(IloInt elt) const`

Inherited Methods from `IloExtractable`
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Constructors

public **IloIntSetVar**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IloIntSetVar**(IloIntSetVarI * impl)

This constructor creates a handle object from a pointer to an implementation object.

public **IloIntSetVar**(const IloEnv env, const IloIntArray array, const char * name=0)

This constructor creates a constrained set variable and makes it part of the environment `env`, where the set consists of integer values. By default, its name is the empty string, but you can specify a `name` of your choice.

public **IloIntSetVar**(const IloEnv env, const IloIntArray possible, const IloIntArray required, const char * name=0)

This constructor creates a constrained set variable and makes it part of the environment `env`, where the set consists of integer values. The array `possible` specifies the set of possible elements of the set variable; the array `required` specifies the set of required elements of the set variable. By default, its name is the empty string, but you can specify a `name` of your choice.

public **IloIntSetVar**(const IloEnv env, const IloNumArray array, const char * name=0)

This constructor creates a constrained set variable and makes it part of the environment `env`, where the set consists of integer values. By default, its name is the empty string, but you can specify a `name` of your choice.

public **IloIntSetVar**(const IloEnv env, const IloNumArray possible, const IloNumArray required, const char * name=0)

This constructor creates a constrained set variable and makes it part of the environment `env`, where the set consists of integer values. The numeric array `possible` specifies the set of possible elements of the set variable; the numeric array `required` specifies the set of required elements of the set variable. By default, its name is the empty string, but you can specify a `name` of your choice.

public **IloIntSetVar**(const IloIntCollection possible, const char * name=0)

This constructor creates a constrained set variable and makes it part of the environment `env`, where the set consists of integer values.

public **IloIntSetVar**(const IloIntCollection possible, const IloIntCollection required, const char * name=0)

This constructor creates a constrained set variable and makes it part of the environment `env`, where the set consists of integer values.

`public IloIntSetVar(const IloNumCollection possible, const char * name=0)`

This constructor creates a constrained set variable and makes it part of the environment `env`, where the set consists of integer values.

`public IloIntSetVar(const IloNumCollection possible, const IloNumCollection required, const char * name=0)`

This constructor creates a constrained set variable and makes it part of the environment `env`, where the set consists of integer values.

Methods

`public void addPossible(IloInt elt) const`

This member function adds `elt` to the set of possible elements of the invoking set variable.

> **Note**
>
> The member function `addPossible` notifies Concert Technology algorithms about this change of this invoking object.

`public void addRequired(IloInt elt) const`

This member function adds `elt` to the set of required elements of the invoking set variable.

> **Note**
>
> The member function `addRequired` notifies Concert Technology algorithms about this change of this invoking object.

`public IloIntSetVarI * getImpl() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public void getPossibleSet(IloIntSet set) const`

This member function accesses the possible elements of the invoking set variable and puts those elements into its argument `set`.

`public IloIntSet getPossibleSet() const`

This member function returns the possible elements of the invoking set variable.

`public void getRequiredSet(IloIntSet set) const`

This member function accesses the possible elements of the invoking set variable and puts those elements into its argument `set`.

`public IloIntSet getRequiredSet() const`

This member function returns the required elements of the invoking set variable.

`public void removePossible(IloInt elt) const`

This member function removes `elt` as a possible element of the invoking set variable.

Note

The member function `removePossible` notifies Concert Technology algorithms about this change of this invoking object.

`public void removeRequired(IloInt elt) const`

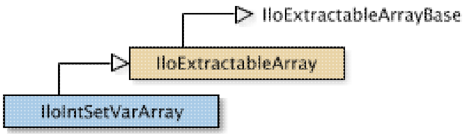
This member function removes `elt` as a required element of the invoking set variable.

Note

The member function `removeRequired` notifies Concert Technology algorithms about this change of this invoking object.

Class IloIntSetVarArray

Definition file: ilconcert/iloset.h



The array class of the set variable class for integer values.
For each basic type, Concert Technology defines a corresponding array class. `IloIntSetVarArray` is the array class of the set variable class for integer values (`IloIntSetVar`) in a model.

Instances of `IloIntSetVarArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added or removed from the array.

Constructor Summary	
public	IloIntSetVarArray(IloDefaultArrayI * i=0)
public	IloIntSetVarArray(const IloEnv env, IloInt n=0)

Method Summary	
public void	add(IloInt more, const IloIntSetVar x)
public void	add(const IloIntSetVar x)
public void	add(const IloIntSetVarArray array)
public IloIntSetVar	operator[](IloInt i) const
public IloIntSetVar &	operator[](IloInt i)

Inherited Methods from `IloExtractableArray`
add, add, add, endElements, setNames

Constructors

`public` **`IloIntSetVarArray`**`(IloDefaultArrayI * i=0)`

This constructor creates an empty extensible array of set variables. You cannot create instances of the undocumented class `IloDefaultArrayI`. As an argument in this default constructor, it allows you to pass 0 (zero) as a value to an optional argument in functions and member functions that accept an array as an argument.

`public` **`IloIntSetVarArray`**`(const IloEnv env, IloInt n=0)`

This constructor creates an extensible array of `n` set variables, where each set is a set of integer values.

Methods

`public void` **`add`**`(IloInt more, const IloIntSetVar x)`

This member function appends `x` to the invoking array multiple times. The argument `more` specifies how many times.

`public void` **`add`**`(const IloIntSetVar x)`

This member function appends `x` to the invoking array.

`public void` **`add`**`(const IloIntSetVarArray array)`

This member function appends the elements in `array` to the invoking array.

`public IloIntSetVar` **`operator[]`**`(IloInt i) const`

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`. On `const` arrays, Concert Technology uses the `const` operator:

```
IloIntSetVar operator[] (IloInt i) const;
```

`public IloIntSetVar &` **`operator[]`**`(IloInt i)`

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`.

Class IloIntTernaryPredicate

Definition file: ilconcert/ilotupleset.h

For constraint programming: ternary predicates operating on arbitrary objects in a model.
This class makes it possible for you to define ternary predicates operating on arbitrary objects in a model. A predicate is an object with a member function (such as `IloIntTernaryPredicate::isTrue`) that checks whether or not a property is satisfied by an ordered set of (pointers to) objects. A ternary predicate checks an ordered set of three objects.

Defining a New Class of Predicates

Predicates, like other Concert Technology objects, depend on two classes: a handle class, `IloIntTernaryPredicate`, and an implementation class, such as `IloIntTernaryPredicateI`, where an object of the handle class contains a data member (the handle pointer) that points to an object (its implementation object) of an instance of `IloIntTernaryPredicateI` allocated in a Concert Technology environment. As a Concert Technology user, you will be working primarily with handles.

If you define a new class of predicates yourself, you must define its implementation class together with the corresponding virtual member function `isTrue`, as well as a member function that returns an instance of the handle class `IloIntTernaryPredicate`.

Arity

As a developer, you can use predicates in Concert Technology applications to define your own constraints that have not already been predefined in Concert Technology. In that case, the *arity* of the predicate (that is, the number of constrained variables involved in the predicate, and thus the size of the array that the member function `IloIntTernaryPredicate::isTrue` must check) must be three.

See Also these classes in the *IBM ILOG Solver Reference Manual*: `IloTableConstraint`.

Constructor and Destructor Summary	
public	IloIntTernaryPredicate()
public	IloIntTernaryPredicate(IloIntTernaryPredicateI * impl)

Method Summary	
public IloIntTernaryPredicateI *	getImpl() const
public IloBool	isTrue(const IloInt val1, const IloInt val2, const IloInt val3)
public void	operator=(const IloIntTernaryPredicate & h)

Constructors and Destructors

`public` **`IloIntTernaryPredicate`**`()`

This constructor creates an empty ternary predicate. In other words, the predicate is an empty handle with a null handle pointer. You must assign the elements of the predicate before you attempt to access it, just as you would any other pointer. An attempt to access it before this assignment will throw an exception (an instance of `IloSolver::SolverErrorException`).

`public` **`IloIntTernaryPredicate`**`(IloIntTernaryPredicateI * impl)`

This constructor creates a handle object (an instance of the class `IloIntTernaryPredicate`) from a pointer to an implementation object (an instance of the implementation class `IloIntTernaryPredicateI`).

Methods

`public IloIntTernaryPredicateI * `**`getImpl`**`() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public IloBool `**`isTrue`**`(const IloInt val1, const IloInt val2, const IloInt val3)`

This member function returns `IloTrue` if the values `val1`, `val2`, and `val3` make the invoking ternary predicate valid. It returns `IloFalse` otherwise.

`public void `**`operator=`**`(const IloIntTernaryPredicate & h)`

This assignment operator copies `h` into the invoking predicate by assigning an address to the handle pointer of the invoking object. That address is the location of the implementation object of the argument `h`. After execution of this operator, both the invoking predicate and `h` point to the same implementation object.

Class IloIntTupleSet

Definition file: ilconcert/ilotupleset.h

Ordered set of values represented by an array.
A tuple is an ordered set of values represented by an array. A *set* of enumerated tuples in a model is represented by an instance of `IloIntTupleSet`. That is, the elements of a tuple *set* are tuples of enumerated values (such as pointers). The number of values in a tuple is known as the *arity* of the tuple, and the arity of the tuples in a set is called the *arity* of the set. (In contrast, the number of tuples in the set is known as the *cardinality* of the set.)

As a handle class, `IloIntTupleSet` manages certain set operations efficiently. In particular, elements can be added to such a set. It is also possible to search a given set with the member function `IloIntTupleSet::isIn` to see whether or not the set contains a given element.

In addition, a set of tuples can represent a constraint defined on a constrained variable, either as the set of *allowed* combinations of values of the constrained variable on which the constraint is defined, or as the set of *forbidden* combinations of values.

There are a few conventions governing tuple sets:

- When you create the set, you must specify the arity of the tuple-elements it contains.
- You use the member function `IloIntTupleSet::add` to add tuples to the set. You can add tuples to the set in a model; you cannot add tuples to an instance of this class during a search, nor inside a constraint, nor inside a goal.

Concert Technology will throw an exception if you attempt:

- to add a tuple with a different number of variables from the arity of the set;
- to search for a tuple with an arity different from the set arity.

See Also: IloIntTupleSetIterator, IloExtractable

Constructor Summary	
`public`	`IloIntTupleSet(const IloEnv env, const IloInt arity)`

Method Summary	
`public IloBool`	`add(const IloIntArray tuple) const`
`public void`	`end()`
`public IloInt`	`getArity() const`
`public IloInt`	`getCardinality() const`
`public IloIntTupleSetI *`	`getImpl() const`
`public IloBool`	`isIn(const IloIntArray tuple) const`
`public IloBool`	`remove(const IloIntArray tuple) const`

Constructors

`public` **`IloIntTupleSet`**`(const IloEnv env, const IloInt arity)`

This constructor creates a set of tuples (an instance of the class `IloIntTupleSet`) with the arity specified by `arity`.

Methods

`public IloBool add(const IloIntArray tuple) const`

This member function adds a tuple represented by the array `tuple` to the invoking set. If you attempt to add an element that is already in the set, that element will *not* be added again. Added elements are not copied; that is, there is no memory duplication. Concert Technology will throw an exception if the size of the array is not equal to the arity of the invoking set. You may use this member function to add tuples to the invoking set in a model; you may not add tuples in this way during a search, inside a constraint, or inside a goal. For those purposes, see `IlcIntTupleSet`, documented in the *IBM ILOG CP Optimizer Reference Manual* and the *IBM ILOG Solver Reference Manual*.

`public void end()`

This member function deletes the invoking set. That is, it frees all the resources used by the invoking object. After a call to this member function, you cannot use the invoking extractable object again.

`public IloInt getArity() const`

This member function returns the arity of the tupleset.

`public IloInt getCardinality() const`

This member function returns the cardinality of the tupleset.

`public IloIntTupleSetI * getImpl() const`

This member function returns a pointer to the implementation object of the invoking extractable object. This member function is useful when you need to be sure that you are using the same copy of the invoking extractable object in more than one situation.

`public IloBool isIn(const IloIntArray tuple) const`

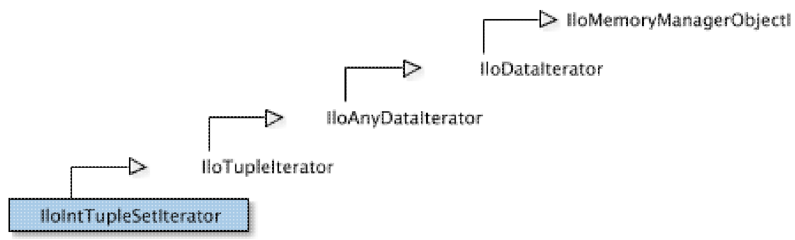
This member function returns `IloTrue` if `tuple` belongs to the invoking set. Otherwise, it returns `IloFalse`. Concert Technology will throw an exception if the size of the array is not equal to the arity of the invoking set.

`public IloBool remove(const IloIntArray tuple) const`

This member function removes `tuple` from the invoking set in a model. You may use this member function to remove tuples from the invoking set in a model; you may not remove tuples in this way during a search, inside a constraint, or inside a goal.

Class IloIntTupleSetIterator

Definition file: ilconcert/ilotupleset.h



Class of iterators to traverse enumerated values of a tuple-set.
An instance of the class `IloIntTupleSetIterator` is an iterator that traverses the elements of a finite set of tuples of enumerated values (instance of `IloIntTupleSet`).

See Also the classes `IlcIntTupleSet` in the *IBM ILOG CP Optimizer Reference Manual* and the *IBM ILOG Solver Reference Manual*.

Constructor Summary	
public	IloIntTupleSetIterator(const IloEnv env, IloIntTupleSet tset)

Method Summary	
public IloIntArray	operator*() const

Constructors

public **IloIntTupleSetIterator**(const IloEnv env, IloIntTupleSet tset)

This constructor creates an iterator associated with `tSet` to traverse its elements.

Methods

public IloIntArray **operator***() const

This operator returns the current element, the one to which the invoking iterator points.

Class IloIntValueSelector

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

Solver lets you create value selectors to control the order in which the values in the domain of a constrained integer variable are tried during the search for a solution.

The class `IloIntValueSelector` represents value selectors in an IBM® ILOG® Concert Technology *model*. The class `IlcIntSelect` represents value selectors internally in a Solver search.

This class is the handle class of the modeling object that wraps the search object. When search starts, `IloIntValueSelectorI` is extracted into an instance of `IlcIntSelectI`.

See Also: IlcIntSelect, IlcIntSelectI, IloIntValueSelectorI

Constructor Summary	
`public`	`IloIntValueSelector()`
`public`	`IloIntValueSelector(IloIntValueSelectorI * impl)`

Method Summary	
`public IloIntValueSelectorI *`	`getImpl() const`
`public void`	`operator=(const IloIntValueSelector & h)`

Constructors

`public` **`IloIntValueSelector`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IloIntValueSelector`**`(IloIntValueSelectorI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

Methods

`public IloIntValueSelectorI *` **`getImpl`**`() const`

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

`public void` **`operator=`**`(const IloIntValueSelector & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IloIntValueSelectorI

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This class is the implementation class for `IloIntValueSelector`.

The class `IloIntValueSelectorI` is the implementation class for value selectors in an IBM® ILOG® Concert Technology *model*. The class `IlcIntSelectI` is the implementation class for value selectors internally in a Solver search.

This class is the modeling object that wraps the search object. When search starts, `IloIntValueSelectorI` is extracted into an instance of `IlcIntSelectI`.

To define new selection criteria, you define both a subclass of `IloIntValueSelectorI` and a subclass of `IlcIntSelectI`.

See Also: IlcIntSelect, IlcIntSelectI, IloIntValueSelector

Constructor and Destructor Summary	
public	IloIntValueSelectorI(IloEnvI *)
public	~IloIntValueSelectorI()

Method Summary	
public virtual void	display(ostream &) const
public virtual IlcIntSelect	extract(const IloSolver solver) const
public IloEnvI *	getEnv() const
public virtual IloIntValueSelectorI *	makeClone(IloEnvI * env) const

Constructors and Destructors

public **IloIntValueSelectorI**(IloEnvI *)

This constructor creates an instance of the class `IloIntValueSelectorI`. This constructor should not be called directly as this class is an abstract class. This constructor is called automatically in the constructor of its subclasses.

public **~IloIntValueSelectorI**()

This destructor is called automatically by the destructor of its subclasses. It frees memory used by the invoking object.

Methods

public virtual void **display**(ostream &) const

This member function prints the invoking value selector on an output stream.

```
public virtual IlcIntSelect extract(const IloSolver solver) const
```

In general terms, in Concert Technology, the objects of a model must be extracted for an algorithm (an instance of one of the subclasses of `IloAlgorithm`, such as `IloSolver`). This member function returns the internal value selector extracted for `solver` from the invoking value selector of a model.

```
public IloEnvI * getEnv() const
```

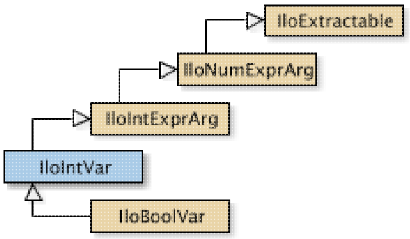
This member function returns the environment to which the invoking value selector belongs. A value selector belongs to exactly one environment; different environments cannot share the same value selector.

```
public virtual IloIntValueSelectorI * makeClone(IloEnvI * env) const
```

This member function is called internally to duplicate the current value selector.

Class IloIntVar

Definition file: ilconcert/iloexpression.h



An instance of this class represents a constrained integer variable in a Concert Technology model.
An instance of this class represents a constrained integer variable in a Concert Technology model. If you are looking for a class of numeric variables that may assume integer values and may be relaxed to assume floating-point values, then consider the class `IloNumVar`. If you are looking for a class of binary decision variables (that is, variables that assume only the values 0 (zero) or 1 (one), then consider the class `IloBoolVar`.

Bounds of an Integer Variable

The lower and upper bound of an instance of this class is an integer.

If you are looking for a symbol to specify an infinite bound, that is, no lower or upper bound, consider `IloIntMin` or `IloIntMax`.

What Is Extracted

An instance of `IloIntVar` is extracted by `IloCP` or `IloSolver` as an instance of `IlcIntVar`.

An instance of `IloIntVar` is extracted by `IloCplex` as a column representing a numeric variable of type `Int` with bounds as specified by `IloIntVar`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

> **Note**
>
> When numeric bounds are given to an integer variable (an instance of `IloIntVar` or `IloNumVar` with `Type = Int`) in the constructors or via a modifier (such as `setUB`, `setLB`, `setBounds`), they are inwardly rounded to an integer value. `LB` is rounded down and `UB` is rounded up.

See Also: IloBoolVar, IloNumVar

Constructor Summary	
`public`	`IloIntVar()`
`public`	`IloIntVar(IloNumVarI * impl)`
`public`	`IloIntVar(IloEnv env, IloInt vmin=0, IloInt vmax=IloIntMax, const char * name=0)`
`public`	`IloIntVar(const IloAddNumVar & var, IloInt lowerBound=0, IloInt upperBound=IloIntMax, const char * name=0)`
`public`	`IloIntVar(const IloEnv env, const IloIntArray values, const char * name=0)`
`public`	`IloIntVar(const IloAddNumVar & var, const IloIntArray values, const char * name=0)`

public	IloIntVar(const IloNumVar var)
public	IloIntVar(const IloIntRange coll, const char * name=0)

Method Summary	
public IloNumVarI *	getImpl() const
public IloNum	getLB() const
public IloInt	getMax() const
public IloInt	getMin() const
public IloNum	getUB() const
public void	setBounds(IloInt lb, IloInt ub) const
public void	setLB(IloNum min) const
public void	setMax(IloInt max) const
public void	setMin(IloInt min) const
public void	setPossibleValues(const IloIntArray values) const
public void	setUB(IloNum max) const

Inherited Methods from `IloIntExprArg`
getImpl

Inherited Methods from `IloNumExprArg`
getImpl

Inherited Methods from `IloExtractable`
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Constructors

public **IloIntVar**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IloIntVar**(IloNumVarI * impl)

This constructor creates a handle object from a pointer to an implementation object.

public **IloIntVar**(IloEnv env, IloInt vmin=0, IloInt vmax=IloIntMax, const char * name=0)

This constructor creates an instance of `IloIntVar` like this:

```
IloNumVar(env, vmin, vmax, ILOINT, name);
```

public **IloIntVar**(const IloAddNumVar & var, IloInt lowerBound=0, IloInt upperBound=IloIntMax, const char * name=0)

This constructor creates an instance of `IloIntVar` like this:

```
IloNumVar(column, lowerBound, upperBound, ILOINT, name);
```

`public IloIntVar(const IloEnv env, const IloIntArray values, const char * name=0)`

This constructor calls upon its corresponding `IloNumVar` constructor.

`public IloIntVar(const IloAddNumVar & var, const IloIntArray values, const char * name=0)`

This constructor calls upon its corresponding `IloNumVar` constructor.

`public IloIntVar(const IloNumVar var)`

This constructor creates a new handle on `var` if it is of type `ILOINT`. Otherwise, an exception is thrown.

`public IloIntVar(const IloIntRange coll, const char * name=0)`

This constructor creates an instance of `IloIntVar` from the given collection

This constructor creates an instance of `IloIntVar` from the given collection.

Methods

`public IloNumVarI * getImpl() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public IloNum getLB() const`

This member function returns the lower bound of the invoking variable.

`public IloInt getMax() const`

This member function returns the maximal value of the invoking variable.

`public IloInt getMin() const`

This member function returns the minimal value of the invoking variable.

`public IloNum getUB() const`

This member function returns the upper bound of the invoking variable.

`public void setBounds(IloInt lb, IloInt ub) const`

This member function sets `lb` as the lower bound and `ub` as the upper bound of the invoking numeric variable.

> **Note**
>
> The member function `setBounds` notifies Concert Technology algorithms about the change of bounds in this numeric variable.

```
public void setLB(IloNum min) const
```

This member function sets `min` as the lower bound of the invoking variable.

> **Note**
>
> The member function `setLB` notifies Concert Technology algorithms about the change of bounds in this numeric variable.

```
public void setMax(IloInt max) const
```

This member function returns the minimal value of the invoking variable to `max`.

> **Note**
>
> The member function `setMax` notifies Concert Technology algorithms about the change of bounds in this numeric variable.

```
public void setMin(IloInt min) const
```

This member function returns the minimal value of the invoking variable to `min`.

> **Note**
>
> The member function `setMin` notifies Concert Technology algorithms about the change of bounds in this numeric variable.

```
public void setPossibleValues(const IloIntArray values) const
```

This member function sets `values` as the domain of the invoking integer variable.

> **Note**
>
> The member function `setPossibleValues` notifies Concert Technology algorithms about the change of bounds in this numeric variable.

```
public void setUB(IloNum max) const
```

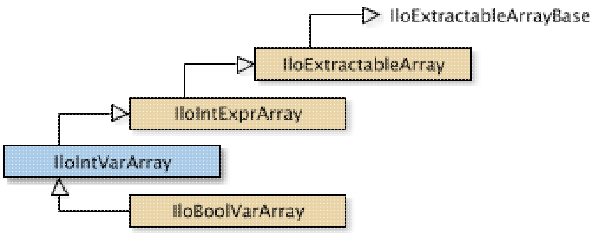
This member function sets `max` as the upper bound of the invoking variable.

> **Note**

The member function `setUB` notifies Concert Technology algorithms about the change of bounds in this numeric variable.

Class IloIntVarArray

Definition file: ilconcert/iloexpression.h



The array class of the integer constrained variables class.
For each basic type, Concert Technology defines a corresponding array class. `IloIntVarArray` is the array class of the integer variable class for a model. It is a handle class.

Instances of `IloIntVarArray` are extensible.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Elements of the array are handles to integer variables. The lower and upper bounds of an integer variable must be an integer. See the documentation of `IloIntVar` for details about bounds of the elements of an array of this class.

See Also: IloIntVar

Constructor Summary	
public	IloIntVarArray(IloDefaultArrayI * i=0)
public	IloIntVarArray(const IloEnv env, IloInt n=0)
public	IloIntVarArray(const IloEnv env, const IloIntArray lb, const IloIntArray ub)
public	IloIntVarArray(const IloEnv env, IloInt lb, const IloIntArray ub)
public	IloIntVarArray(const IloEnv env, const IloIntArray lb, IloInt ub)
public	IloIntVarArray(const IloEnv env, IloInt n, IloInt lb, IloInt ub)
public	IloIntVarArray(const IloEnv env, const IloNumColumnArray columnarray)
public	IloIntVarArray(const IloEnv env, const IloNumColumnArray columnarray, const IloNumArray lb, const IloNumArray ub)

Method Summary	
public void	add(IloInt more, const IloIntVar x)
public void	add(const IloIntVar x)
public void	add(const IloIntVarArray x)
public void	endElements()
public IloIntVar	operator[](IloInt i) const
public IloIntVar &	operator[](IloInt i)
public IloIntExprArg	operator[](IloIntExprArg anIntegerExpr) const
public IloNumVarArray	toNumVarArray() const

Inherited Methods from `IloIntExprArray`
`add, add, add, endElements, operator[], operator[], operator[]`

Inherited Methods from `IloExtractableArray`
`add, add, add, endElements, setNames`

Constructors

`public IloIntVarArray(IloDefaultArrayI * i=0)`

This constructor creates an empty extensible array of integer variables.

`public IloIntVarArray(const IloEnv env, IloInt n=0)`

This constructor creates an extensible array of `n` integer variables.

`public IloIntVarArray(const IloEnv env, const IloIntArray lb, const IloIntArray ub)`

This constructor creates an extensible array of integer variables with lower and upper bounds as specified.

`public IloIntVarArray(const IloEnv env, IloInt lb, const IloIntArray ub)`

This constructor creates an extensible array of integer variables with a lower bound and an array of upper bounds as specified.

`public IloIntVarArray(const IloEnv env, const IloIntArray lb, IloInt ub)`

This constructor creates an extensible array of integer variables with an array of lower bounds and an upper bound as specified.

`public IloIntVarArray(const IloEnv env, IloInt n, IloInt lb, IloInt ub)`

This constructor creates an extensible array of `n` integer variables, with a lower and an upper bound as specified.

`public IloIntVarArray(const IloEnv env, const IloNumColumnArray columnarray)`

This constructor creates an extensible array of integer variables from a column array.

`public IloIntVarArray(const IloEnv env, const IloNumColumnArray columnarray, const IloNumArray lb, const IloNumArray ub)`

This constructor creates an extensible array of integer variables with lower and upper bounds as specified from a column array.

Methods

`public void add(IloInt more, const IloIntVar x)`

This member function appends `x` to the invoking array of integer variables; it appends `x` `more` times.

`public void add(const IloIntVar x)`

This member function appends the value `x` to the invoking array.

`public void add(const IloIntVarArray x)`

This member function appends the variables in the array `x` to the invoking array.

`public void endElements()`

This member function calls `IloExtractable::end` for each of the elements in the invoking array. This deletes all the extractables identified by the array, leaving the handles in the array intact. This member function is the recommended way to delete the elements of an array.

`public IloIntVar operator[](IloInt i) const`

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`. On `const` arrays, Concert Technology uses the `const` operator

```
IloIntVar operator[] (IloInt i) const;
```

`public IloIntVar & operator[](IloInt i)`

This operator returns a reference to the extractable object located in the invoking array at the position specified by the index `i`.

`public IloIntExprArg operator[](IloIntExprArg anIntegerExpr) const`

This subscripting operator returns an expression argument for use in a constraint or expression. For clarity, let's call `A` the invoking array. When `anIntegerExpr` is bound to the value `i`, the domain of the expression is the domain of `A[i]`. More generally, the domain of the expression is the union of the domains of the expressions `A[i]` where the `i` are in the domain of `anIntegerExpr`.

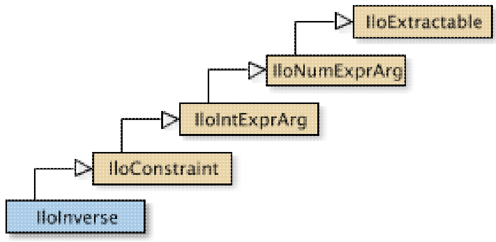
This operator is also known as an element expression.

`public IloNumVarArray toNumVarArray() const`

This member function copies the invoking array into a new `IloNumVarArray`.

Class IloInverse

Definition file: ilconcert/ilomodel.h



For constraint programming: constrains elements of one array to be inverses of another.
An instance of `IloInverse` represents an inverse constraint. Informally, we say that an inverse constraint works on two arrays, say, `f` and `invf`, so that an element of `f` composed with the corresponding element of `invf` produces the index of that element.

In formal terms, if the length of the array `f` is `n`, and the length of the array `invf` is `m`, then the inverse constraint makes sure that:

- for all `i` in the interval `[0, n-1]`, if `f[i]` is in `[0, m-1]` then `invf[f[i]] == i`;
- for all `j` in the interval `[0, m-1]`, if `invf[j]` is in `[0, n-1]` then `f[invf[j]] == j`.

In order for a constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloConstraint

Constructor Summary	
public	`IloInverse()`
public	`IloInverse(IloInverseI * impl)`
public	`IloInverse(const IloEnv env, const IloIntVarArray f, const IloIntVarArray invf, const char * name=0)`

Method Summary	
public IloInverseI *	`getImpl() const`

Inherited Methods from `IloConstraint`
getImpl

Inherited Methods from `IloIntExprArg`
getImpl

Inherited Methods from `IloNumExprArg`
getImpl

Inherited Methods from `IloExtractable`

```
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv,
getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr,
isObjective, isVariable, setName, setObject
```

Constructors

`public IloInverse()`

This constructor creates an empty handle. You must initialize it before you use it.

`public IloInverse(IloInverseI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public IloInverse(const IloEnv env, const IloIntVarArray f, const IloIntVarArray invf, const char * name=0)`

This constructor creates an inverse constraint that if the length of the array `f` is `n`, and the length of the array `invf` is `m`, then this function returns a constraint that insures that:

- for all `i` in the interval `[0, n-1]`, if `f[i]` is in `[0, m-1]` then `invf[f[i]] == i`;
- for all `j` in the interval `[0, m-1]`, if `invf[j]` is in `[0, n-1]` then `f[invf[j]] == j`.

Methods

`public IloInverseI * getImpl() const`

This member function returns a pointer to the implementation object of the invoking handle.

Class IloIterator<>

Definition file: ilconcert/iloiterator.h

A template to create iterators for a class of extractable objects.
This template creates iterators for a given class of extractable objects (denoted by `E` in the template) within an instance of `IloEnv`.

By default, an iterator created in this way will traverse instances of `E` and of its subclasses. You can prevent the iterator from traversing instances of subclasses of E (that is, you can limit its effect) by setting the argument `withSubClasses` to `IloFalse` in the constructor of the iterator.

While an iterator created in this way is working, you must not create nor destroy any extractable objects in the instance of `IloEnv` where it is working. In other words, an iterator created in this way works only in a stable environment.

An iterator created with this template differs from an instance of `IloModel::Iterator`. An instance of `IloModel::Iterator` works only on extractable objects (instances of `IloExtractable` or its subclasses) that have explicitly been added to a model (an instance of `IloModel`). In contrast, an iterator created with this template will work on all extractable objects within a given environment, whether or not they have been explicitly added to a model.

See Also: IloEnv, IloExtractable, IloModel, IloModel::Iterator

Constructor Summary	
public	IloIterator(const IloEnv env, IloBool withSubClasses=IloTrue)

Method Summary	
public IloBool	ok()
public void	operator++()

Constructors

public **IloIterator**(const IloEnv env, IloBool withSubClasses=IloTrue)

This template constructor creates an iterator for instances of the class `E`. When the argument `withSubClasses` is `IloTrue` (its default value), the iterator will also work on instances of the subclasses of `E`. When `withSubClasses` is `IloFalse`, the iterator works only on instances of `E`.

Example

Here is an example of an iterator created by this template for the class IloNumVar.

```
typedef IloIterator<IloNumVar> IloNumVarIterator;

void displayAllVars(IloEnv env) {
  for (IloNumVarIterator it(env); it.ok(); ++it) {
    IloNumVar ext = *it;
    cout << ext;
  }
}
```

Methods

`public IloBool ok()`

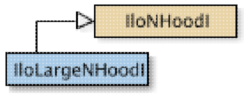
This member function returns `IloTrue` if there is a current element and the iterator points to it. Otherwise, it returns `IloFalse`.

`public void operator++()`

This operator advances the iterator to point to the next value in the iteration.

Class IloLargeNHoodI

Definition file: ilsolver/iimlns.h
Include file: <ilsolver/iimlns.h>



Special neighborhood implementation for Large Neighborhood Search.
This class is a special sub-class of `IloNHoodI` which is designed to ease the writing of Large Neighborhood Search methods (see the *Solver User's Manual*). Large Neighborhood Search is a local search technique which relies on constraint programming to explore a neighborhood which is defined by allowing a subset of the decision variables to change their value from that taken in the current solution.

Traditional Solver goals are used to perform the exploration of the neighborhood, but the `IloNHoodI` class is used to define exactly what variables may change their value. This variable subset is usually referred to as the *solution fragment.* Technically, the way in which this fragment is created is to return a *solution delta* (`IloNHoodI::define`) from `IloNHoodI::define` which includes all the variables of the fragment, but with a non-restorable status (see `IloSolution::setRestorable`). This will induce the Solver's local search to produce a neighbor which has all the variables in the fragment in their uninstantiated state, their domains reduced only by the propagation of problem constraints. That is, it is as if the variables mentioned in the delta had their current assignments relaxed. At this point a *completion goal* can explore all (or some) combinations of values for these variables, in an attempt to look for a better solution.

This class makes it possible to define deltas as described above without worrying about restore status and so on. One of the easiest ways to subclass this class is to use the macro `ILODEFINELNSFRAGMENT0` (or one of its variants) which is ideal for most uses. Direct subclassing is seldom necessary, except for more advanced uses. Whether you use `ILODEFINELNSFRAGMENT0`, or directly subclass `IloLargeNHoodI`, the only thing you really need to worry about is that you call the `IloLargeNHoodI::addToFragment` member function for each variable you wish to be able to change its value.

You can operate this class in essentially two modes. The first is the simplest and is used by `ILODEFINELNSFRAGMENT0`. This mode supposes that the neighborhood has only one neighbor (which might be termed a meta-neighbor as it creates the possibility to move to a large number of distinct assignments) and the solution fragment is defined by the member function `IloLargeNHoodI::defineFragment`. Diversity of search is ensured by generating the fragment in a randomized way. In this case, `IloLargeNHoodI::defineFragment` is the only function that you need define in the subclass.

In the second mode, the neighborhood can have an arbitrary number of neighbors. Here, diversity is assured via the number of neighbors (as for traditional neighborhoods), and so randomization is not essential (but also not forbidden). In this second case, two member functions need to be redefined: `IloLargeNHoodI::getSize`, which delivers the number of neighbors, and `IloLargeNHoodI::defineFragmentAtIndex` which defines the fragment for a particular index given. As an example of this second kind of neighborhood, consider fragments defined over an array of variables *x* of size *n*. If there is some interest in relaxing a contiguous portion of the variables, you might consider as fragment *i* the variables of indices *i-k* to *i+k* (ignoring boundary conditions for now) where *k* is a non-negative integer.

Note

This class has special versions of the member functions `IloLargeNHoodI::start` and `IloLargeNHoodI::define`. If you must overload one or both of these, you must ensure that you call `IloLargeNHoodI::start` or `IloLargeNHoodI::define` as appropriate in your sub-class.

See Also: ILODEFINELNSFRAGMENT0, IloNHoodI, IloSingleMove, IlcRandom

Constructor and Destructor Summary	
`public`	`IloLargeNHoodI(IloEnv env, const char * name=0)`

	Creates a neighborhood for Large Neighborhood Search.

Method Summary	
public void	addToFragment(IloSolver solver, IloAnySetVar var)
public void	addToFragment(IloSolver solver, IloIntSetVar var)
public void	addToFragment(IloSolver solver, IloAnyVar var)
public void	addToFragment(IloSolver solver, IloNumVar var)
public void	addToFragment(IloSolver solver, IloIntVar var) Adds a variable to the fragment being defined.
public IloSolution	define(IloSolver solver, IloInt index) Gets a meta-neighbor from the large neighborhood.
public virtual void	defineFragment(IloSolver solver) Defines a Large Neighborhood Search fragment.
public virtual void	defineFragmentAtIndex(IloSolver solver, IloInt index) Defines a Large Neighborhood Search fragment.
public IloSolution	getCurrentSolution() const Returns the current solution passed to `IloLargeNHoodI::start`.
public IloInt	getSize(IloSolver solver) const Delivers the size (number of neighbors) of the neighborhood.
public IloBool	isInFragment(IloSolver solver, IloExtractable var) const Indicates if a particular variable is in the fragment.
public void	start(IloSolver solver, IloSolution solution) Starts the large neighborhood.

Inherited Methods from `IloNHoodI`
define, display, getEnv, getLocalIndex, getLocalNHood, getName, getObject, getSize, notify, notifyOther, operator delete, reset, setName, setObject, start

Constructors and Destructors

`public` **`IloLargeNHoodI`**`(IloEnv env, const char * name=0)`

Creates a neighborhood for Large Neighborhood Search.

This constructor creates an instance of a neighborhood to be used with Large Neighborhood Search on an environment, using `env` as an allocation environment. The optional name `name` becomes the name of the newly created neighborhood.

Methods

`public void` **`addToFragment`**`(IloSolver solver, IloIntVar var)`
`public void` **`addToFragment`**`(IloSolver solver, IloAnySetVar var)`
`public void` **`addToFragment`**`(IloSolver solver, IloIntSetVar var)`
`public void` **`addToFragment`**`(IloSolver solver, IloAnyVar var)`

`public void` **`addToFragment`**`(IloSolver solver, IloNumVar var)`

Adds a variable to the fragment being defined.

This member function should be called from either `IloLargeNHoodI::defineFragment` or `IloLargeNHoodI::defineFragmentAtIndex` (depending on which one is overloaded) to add a variable to the fragment currently being defined. You should pass as `solver` the instance of the Solver received in `IloLargeNHoodI::defineFragment` or `IloLargeNHoodI::defineFragmentAtIndex`. `var` is the variable to add to the fragment. If `var` is already in the fragment, then an exception (an instance of `IloException`) is raised. The member function `IloLargeNHoodI::isInFragment` can be used to determine if a variable is already in the fragment.

`public IloSolution` **`define`**`(IloSolver solver, IloInt index)`

Gets a meta-neighbor from the large neighborhood.

Normally, you need not overload this member function. It performs some setup tasks, calls `IloLargeNHoodI::defineFragmentAtIndex`, and afterwards builds the solution delta from a record of the calls that were made to `IloLargeNHoodI::addToFragment`. It is strongly recommended that you do not overload this function. If overloading is necessary, you must call `IloLargeNHoodI::define(solver, index)` and return the defined solution delta.

`public virtual void` **`defineFragment`**`(IloSolver solver)`

Defines a Large Neighborhood Search fragment.

When you wish to create a large neighborhood which will use a randomized procedure to generate diverse fragments, this is the member function you should overload. In this member function, you make a call to `IloLargeNHoodI::addToFragment` for each variable you wish to add to the fragment. The default behavior of this function is to raise an exception (an instance of `IloException`) indicating that a fragment-defining function must be overloaded.

See Also: ILODEFINELNSFRAGMENT0

`public virtual void` **`defineFragmentAtIndex`**`(IloSolver solver, IloInt index)`

Defines a Large Neighborhood Search fragment.

When you wish to create a large neighborhood which will create a distinct set of fragments, this is the member function (together with `IloLargeNHoodI::getSize`) that you should overload. The default behavior is to call `IloLargeNHoodI::defineFragment`, ignoring the index `index`.

`public IloSolution` **`getCurrentSolution`**`() const`

Returns the current solution passed to `IloLargeNHoodI::start`.

This member function returns the current solution from which fragments are to be defined. This solution was passed to the `IloLargeNHoodI::start` function to begin looking for neighbors.

See Also: IloNHoodI::start

`public IloInt` **`getSize`**`(IloSolver solver) const`

Delivers the size (number of neighbors) of the neighborhood.

This member function should be overloaded if you wish to define a Large Neighborhood Search with more than a single meta-neighbor. This function should return the number of meta-neighbors (ways of defining a fragment).

When this function is overloaded, you should also overload the `IloLargeNHoodI::defineFragmentAtIndex` to return the correct fragment for any given index.

`public IloBool` **`isInFragment`**`(IloSolver solver, IloExtractable var) const`

Indicates if a particular variable is in the fragment.

This member function returns `IloTrue` if and only if the variable `var` is in the fragment. Otherwise, it returns `IloFalse`. The parameter `solver` is the instance of `IloSolver` driving the neighborhood.

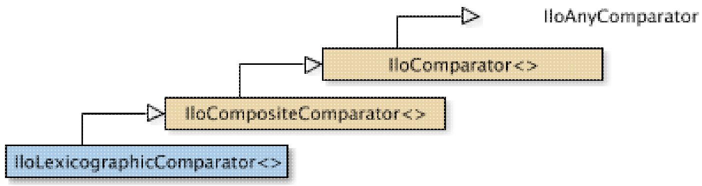
`public void` **`start`**`(IloSolver solver, IloSolution solution)`

Starts the large neighborhood.

Normally, you need not overload this member function. It performs some housekeeping tasks, and keeps `solution` locally so that you can access it via `IloLargeNHoodI::getCurrentSolution`. If you do need to overload this function, ensure that you call `IloLargeNHoodI::start(solver, solution)` before any other code in your function.

Class IloLexicographicComparator<>

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>



This class composes comparators lexicographically.
A lexicographic comparator is a composite comparator, `CC`, made of an ordered set `O`, of comparators `ci`. The result of comparing two objects `o1` and `o2` with `CC`, denoted `CC(o1, o2)`, is as follows:

```
CC(o1, o2) = 0
       if for all i, ci(o1, o2) = 0, ci element O

CC(o1, o2) = ci(o1, o2)
     if ci(o1, o2) != 0 AND
       for all j < i cj(o1, o2) = 0,
        ci, cj element O
```

The function `IloCompositeComparator::add` is available to build a comparator by addition rather than by specifying all comparators at construction time.

For more information, see Selectors.

See Also: IloComposeLexical

Constructor Summary	
`public`	`IloLexicographicComparator(IloMemoryManager manager)` Initializes an empty lexicographic comparator.

Inherited Methods from `IloCompositeComparator`
`add`

Inherited Methods from `IloComparator`
`isBetterOrEqual, isBetterThan, isEqual, isWorseOrEqual, isWorseThan, makeInverse, operator()`

Constructors

`public` **`IloLexicographicComparator`**`(IloMemoryManager manager)`

Initializes an empty lexicographic comparator.

This constructor intializes an empty lexicographic comparator allocated on the memory manager `manager`.

Class IloCsvReader::IloLineNotFoundException

Definition file: ilconcert/ilocsvreader.h

Exception thrown for unfound line.
This exception is thrown by the following member functions if the line is not found.

- IloCsvTableReader::getLineByKey
- IloCsvTableReader::getLineByNumber
- IloCsvReader::getLineByKey
- IloCsvReader::getLineByNumber

Class IloListener

Definition file: ilsolver/iimevent.h
Include file: <ilsolver/iim.h>

The listener class.
Listeners are `IloEvent` handlers. You can easily define new listeners by using the `ILOIIMLISTENER0` macro or one of its variants.

See Also: ILOIIMLISTENER0

Method Summary	
public void	end()
public IloEnv	getEnv() const

Methods

`public void` **`end`**`()`

brief Destroys the object.

This function deletes the object from the environment on which it was allocated and sets the implementation pointer to zero.

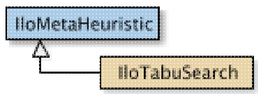
`public IloEnv` **`getEnv`**`() const`

brief Returns the allocation environment.

This function returns the environment on which the invoking object was allocated.

Class IloMetaHeuristic

Definition file: ilsolver/iimmeta.h
Include file: <ilsolver/iimls.h>

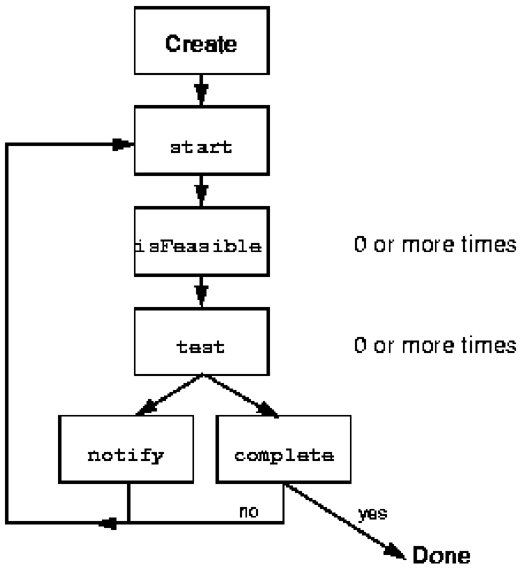


This handle class comprises methods that describe a metaheuristic that can be used within local search procedures. The metaheuristic defines a filter for the set of possible moves that can be taken, and thus provides search guidance.

The exact choice of which legal move is to be taken is performed by search selectors (objects of type `IloSearchSelector`), not by the `IloMetaHeuristic` class.

Protocol

When used in a local search framework, metaheuristics should be called according to the following protocol:



`IloSingleMove` adheres to this protocol.

See Also: IloApplyMetaHeuristic, IloMetaHeuristicI, IloNotify, IloScanDeltas, IloScanNHood, IloSingleMove, IloStart, IloTest

Constructor Summary	
`public`	`IloMetaHeuristic()`
`public`	`IloMetaHeuristic(IloMetaHeuristicI * impl)`

Method Summary	
`public IloBool`	`complete()`
`public void`	`end(IloBool deep=IloTrue)`
`public IloSolutionDeltaCheck`	`getDeltaCheck() const`

public IloEnv	getEnv()
public IloMetaHeuristicI *	getImpl() const
public const char *	getName() const
public IloAny	getObject() const
public IloBool	isFeasible(IloSolver solver, IloSolution delta) const
public void	notify(IloSolver solver, IloSolution delta)
public void	operator=(const IloMetaHeuristic & mh)
public void	reset()
public void	setName(const char * name) const
public void	setObject(IloAny obj) const
public IloBool	start(IloSolver solver, IloSolution solution)
public IloBool	test(IloSolver solver, IloSolution delta)

Constructors

`public` **`IloMetaHeuristic`**`()`

This constructor creates a metaheuristic whose handle pointer is null. This object must be assigned before it can be used.

`public` **`IloMetaHeuristic`**`(IloMetaHeuristicI * impl)`

This constructor creates a handle object (an instance of `IloMetaHeuristic`) from a pointer to an implementation object (an instance of `IloMetaHeuristicI`).

Methods

`public IloBool` **`complete`**`()`

This member function is normally called when the search has stagnated (typically when no neighborhood moves can be legally taken). This allows the metaheuristic to perform some action to allow the search to continue, if desired. If the return value is `IloTrue`, the metaheuristic wants to end local search at this point.

`public void` **`end`**`(IloBool deep=IloTrue)`

This member function calls `operator delete` from the class `IloMetaHeuristicI` on the implementation object associated with the invoking handle and sets the implementation pointer to 0. If the invoking metaheuristic has any children (for instance, if it were constructed using a `+` operator) then these too will be likewise deleted. If however, `deep` has the value `IloFalse`, then the children are “disconnected” from the parent before `delete` is called on the implementation pointer to ensure that the children of the invoking metaheuristic are not destroyed.

`public IloSolutionDeltaCheck` **`getDeltaCheck`**`() const`

Meta-heuristics are capable of performing “prefiltering” of solution deltas to be applied to a solution.This member function returns an object that performs the pre-filtering for the metaheuristic. This object may be passed to

`IloScanNHood`.

`public IloEnv` **`getEnv`**`()`

This member function returns the environment associated with the implementation class.

`public IloMetaHeuristicI *` **`getImpl`**`() const`

This member function returns the implementation object of the invoking metaheuristic (a handle). You can use this member function to check whether a metaheuristic is empty.

`public const char *` **`getName`**`() const`

This member function returns a character string specifying the name of the invoking object (if there is one).

`public IloAny` **`getObject`**`() const`

This member function returns the object associated with the invoking object (if there is one). Normally, an associated object contains user data pertinent to the invoking object.

`public IloBool` **`isFeasible`**`(IloSolver solver, IloSolution delta) const`

This member function is called by the `ok()` method of the object returned by `IloMetaHeuristic::getDeltaCheck`. That object is used in local search procedures to “pre-filter” some deltas before they are applied to the current solution.

The argument `solver` indicates the solver that is currently performing a local search using the invoking metaheuristic.

The user can define this member function to perform the pre-filtering desired. The default behavior is to return `IloTrue`, indicating that `delta` is feasible.

`public void` **`notify`**`(IloSolver solver, IloSolution delta)`

Call this member function when a neighborhood move is to be taken by local search. At that point, all solution variables should be instantiated, but not yet saved, in the current solution, thus allowing differences to be calculated. This can be useful for updating metaheuristic data structures.

The argument `solver` indicates the solver that is currently performing a local search using the invoking metaheuristic.

The argument `delta` holds the solution delta of the move to be taken. This can be gleaned from the reference parameter `atDelta` passed to `IloScanDeltas` or `IloScanNHood`, and used in `IloNotify`. While this parameter can be used in custom metaheuristics, an empty handle can be passed to all of Solver's pre-defined metaheuristics, as it is not required by them.

`public void` **`operator=`**`(const IloMetaHeuristic & mh)`

This assignment operator copies `mh` into the invoking metaheuristic by assigning an address to the handle pointer of the invoking object. That address is the location of the implementation object of the argument `mh`.

```
public void reset()
```

This member function resets the metaheuristic to the state in which it was first created. When metaheuristics have state, you will typically want to call this member function between two different local searches if the metaheuristic is to be reused. For metaheuristics which have no state, calling this member function has no effect. (All of Solver's metaheuristics have state except `IloImprove`.)

```
public void setName(const char * name) const
```

This member function assigns `name` to the invoking object.

```
public void setObject(IloAny obj) const
```

This member function associates `obj` with the invoking object. The member function `getObject` accesses this associated object afterward. Normally, `obj` contains user data pertinent to the invoking object.

```
public IloBool start(IloSolver solver, IloSolution solution)
```

Call this member function when a local search begins to search for a new move to take. The argument `solver` indicates the solver that is currently performing a local search using the invoking metaheuristic. The argument `solution` is the current solution. If it is illegal for the metaheuristic to start from this point, this member function returns `IloFalse` or causes a failure. Otherwise it returns `IloTrue`.

```
public IloBool test(IloSolver solver, IloSolution delta)
```

When all solution variables have been instantiated in local search, call this member function to determine whether the metaheuristic should allow or reject the solution. If the solution is rejected, this member function returns `IloFalse` or causes a failure. Otherwise it returns `IloTrue`.

The argument `solver` indicates the solver that is currently performing a local search using the invoking metaheuristic.

The argument `delta` holds the solution delta to test. This can be gleaned from the reference parameter `atDelta` passed to `IloScanDeltas` or `IloScanNHood`, and used in `IloTest`. While this parameter can be used in custom metaheuristics, an empty handle can be passed to all of Solver's pre-defined metaheuristics, as it is not required by them.

Class IloMetaHeuristicI

Definition file: ilsolver/iimmeta.h
Include file: <ilsolver/iimls.h>

This implementation class comprises methods that allow you to define your own a metaheuristic which can then be used within local search procedures. When you define a metaheuristic, you define a filtering of the set of possible moves that can be taken, and thus can provide search guidance.

The member functions `IloMetaHeuristicI::isFeasible`, `IloMetaHeuristicI::notify`, `IloMetaHeuristicI::start`, and `IloMetaHeuristicI::test` are called within search, and the solver calling these is passed as the first parameter of these member functions.

See Also: IloApplyMetaHeuristic, IloMetaHeuristic, IloNotify, IloScanDeltas, IloScanNHood, IloSingleMove, IloStart, IloTest

Constructor and Destructor Summary	
public	IloMetaHeuristicI(IloEnv env, const char * name=0)
public	~IloMetaHeuristicI()

Method Summary	
public virtual IloBool	complete()
public IloSolutionDeltaCheck	getDeltaCheck() const
public char *	getName() const
public IloAny	getObject()
public virtual IloBool	isFeasible(IloSolver solver, IloSolution delta) const
public virtual void	notify(IloSolver solver, IloSolution delta)
public void	operator delete(void * p, size_t size)
public virtual void	reset()
public void	setName(const char * name)
public void	setObject(IloAny object)
public virtual IloBool	start(IloSolver solver, IloSolution solution)
public virtual IloBool	test(IloSolver solver, IloSolution delta)

Constructors and Destructors

`public` **`IloMetaHeuristicI`**`(IloEnv env, const char * name=0)`

This constructor creates an instance of `IloMetaHeuristicI` associated with the environment `env`. The optional parameter `name`, if supplied, becomes the name of the metaheuristic.

`public` **`~IloMetaHeuristicI`**`()`

Since `IloMetaHeuristicI` is an abstract class that can be sub-classed, a virtual destructor is provided.

Methods

`public virtual IloBool complete()`

This member should be called by local search procedures when the search has stagnated (typically when no neighborhood moves can be legally taken). This allows the metaheuristic to perform some action to allow the search to continue, if desired. If the return value is `IloTrue` (the default behavior), the metaheuristic wants to end local search at this point.

`public IloSolutionDeltaCheck getDeltaCheck() const`

Metaheuristics are capable of performing "pre-filtering" of solution deltas to be applied to a solution. This member function returns an object that performs the pre-filtering for the metaheuristic.

The implementation of the `IloSolutionDeltaCheckI::ok` method for this object is to call the `IloMetaHeuristicI::isFeasible` method in the invoking object with the proposed delta. Thus, the behavior of the delta check here is defined by the behavior of the `IloMetaHeuristicI::isFeasible` method.

`public char * getName() const`

This member function returns the name specified in the constructor of the object, or specified in the last call to `IloMetaHeuristicI::setName`.

`public IloAny getObject()`

This member function returns the object associated with the invoking neighborhood through the `IloMetaHeuristicI::setObject` method.

`public virtual IloBool isFeasible(IloSolver solver, IloSolution delta) const`

This member function is called by the `ok()` method of the object returned by `IloMetaHeuristic::getDeltaCheck`. That object is used in local search procedures to “pre-filter” some deltas before they are applied to the current solution.

The argument `solver` indicates the solver that is currently performing a local search using the invoking metaheuristic.

The user can define this member function to perform the pre-filtering desired. The default behavior is to return `IloTrue` to that the `delta` is feasible.

`public virtual void notify(IloSolver solver, IloSolution delta)`

Call this member function when a neighborhood move is to be taken by local search. At that point, all solution variables should be instantiated, but not yet saved, in the current solution, thus allowing differences to be calculated. This can be useful for updating metaheuristic data structures. The default behavior is to do nothing.

The argument `solver` indicates the solver that is currently performing a local search using the invoking metaheuristic.

The argument `delta` holds the solution delta of the move to be taken. This information is not absolutely necessary, as variable changes can be worked out from an examination of the variables themselves. Thus, `delta` may be an empty handle, for example when this method is called from the `IloNotify` goal without the `delta` parameter. However, when `IloSingleMove` is used, `delta` always represents the delta of the move to be taken.

If `delta` is not empty, the information it contains can be used to accelerate the notification process for some metaheuristics or to hold additional information on the move.

```
public void operator delete(void * p, size_t size)
```

This operator deletes the memory for the metaheuristic pointed to by `p` as if its memory was allocated using the environment handed to the metaheuristic when it was created.

If you want to allocate your metaheuristics on a different heap when subclassing, you should redefine this operator in the subclass to perform the delete operation appropriate to your heap.

```
public virtual void reset()
```

You should define this member function to reset the metaheuristic to the state in which it was first created, unless your metaheuristic does not have state which it carries from move to move. (All Solver metaheuristics have state except `IloImprove`.)

```
public void setName(const char * name)
```

This member function sets the name of the invoking object to a copy of `name`.

```
public void setObject(IloAny object)
```

This member function associates object `object` with the invoking metaheuristic.

```
public virtual IloBool start(IloSolver solver, IloSolution solution)
```

This member function should be called when local search begins to search for a new move to take. The argument `solver` indicates the solver that is currently performing a local search using the invoking metaheuristic. The argument `solution` is the current solution. If it is illegal for the metaheuristic to start from this point, this member function returns `IloFalse` or causes a failure. Otherwise it returns `IloTrue`.

```
public virtual IloBool test(IloSolver solver, IloSolution delta)
```

This member function is the principal way in which the metaheuristic guides search by filtering neighbors.

This member function is called when all solution variables have been instantiated in local search, to determine whether the metaheuristic should allow or reject the solution. If the solution is rejected, this member function returns `IloFalse` or causes a failure. Otherwise it returns `IloTrue`.

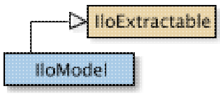
The argument `solver` indicates the solver that is currently performing a local search using the invoking metaheuristic.

The argument `delta` holds the solution delta to test. This information is not absolutely necessary, as variable changes can be worked out from an examination of the variables themselves. Thus, `delta` may be an empty handle, for example when this method is called from the `IloTest` goal without the delta parameter. However, when `IloSingleMove` is called, `delta` always represents the delta to be tested.

If `delta` is not empty, the information it contains can be used to accelerate the notification process for some metaheuristics or to hold additional information on the move.

Class IloModel

Definition file: ilconcert/ilomodel.h



Class for models.
An instance of this class represents a model. A model consists of the extractable objects such as constraints, constrained variables, objectives, and possibly other modeling objects, that represent a problem. Concert Technology extracts information from a model and passes the information in an appropriate form to algorithms that solve the problem. (For information about extracting objects into algorithms, see the member function `IloAlgorithm::extract` and the template `IloAdd`.)

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Models and Submodels

With Concert Technology, you may create more than one model in a given environment (an instance of `IloEnv`). In fact, you can create submodels. That is, you can add one model to another model within the same environment.

What Is Extracted from a Model

All the extractable objects (that is, instances of `IloExtractable` or one of its subclasses) that have been added to a model (an instance of `IloModel`) and that have not been removed from it will be extracted when an algorithm extracts the model. An instance of the nested class `IloModel::Iterator` accesses those extractable objects.

See Also: IloEnv, IloExtractable, IloModel::Iterator

Constructor Summary	
`public`	`IloModel()`
`public`	`IloModel(IloModelI * impl)`
`public`	`IloModel(const IloEnv env, const char * name=0)`

Method Summary	
`public const IloExtractableArray &`	`add(const IloExtractableArray & x) const`
`public IloExtractable`	`add(const IloExtractable x) const`
`public IloModelI *`	`getImpl() const`
`public void`	`remove(const IloExtractableArray x) const`
`public void`	`remove(const IloExtractable x) const`

Inherited Methods from `IloExtractable`
`asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject`

Inner Class	
IloModel::Iterator	Nested class of iterators to traverse the extractable objects in a model.

Constructors

`public IloModel()`

This constructor creates an empty handle. You must initialize it before you use it.

`public IloModel(IloModelI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public IloModel(const IloEnv env, const char * name=0)`

This constructor creates a model. By default, the name of the model is the empty string, but you can attribute a name to the model at its creation.

Methods

`public const IloExtractableArray & add(const IloExtractableArray & x) const`

This member function adds the array of extractable objects to the invoking model.

> **Note**
>
> The member function `add` notifies Concert Technology algorithms about this addition to the model.

`public IloExtractable add(const IloExtractable x) const`

This member function adds the extractable object to the invoking model.

> **Note**
>
> The member function `add` notifies Concert Technology algorithms about this addition to the model.

`public IloModelI * getImpl() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public void remove(const IloExtractableArray x) const`

This member function removes the array of extractable objects from the invoking model.

> **Note**
>
> The member function `remove` notifies Concert Technology algorithms about this removal from the model.

`public void remove(const IloExtractable x) const`

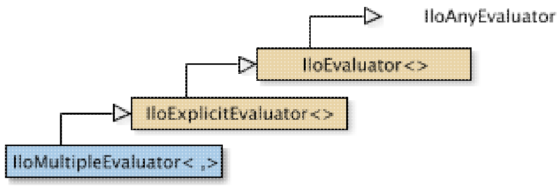
This member function removes the extractable object from the invoking model.

Note

The member function `remove` notifies Concert Technology algorithms about this removal from the model.

Class IloMultipleEvaluator<,>

Definition file: ilsolver/iimmulti.h
Include file: <ilsolver/iim.h>



An explicit evaluator which can be refreshed from a container class.
This class is a simple extension to the `IloExplicitEvaluator` class, which allows the explicit evaluator to be filled with the objects (and their evaluations) found in a container class.

This class is best used when a particular container changes relatively infrequently, but at known times. When this happens, you can update the evaluator explicitly by asking the evaluator to take evaluations from the container.

This class can be created in two ways. The first way of creating an instance of this class is from a *subordinate evaluator*, so that when the update from the container happens, this subordinate evaluator is used to evaluate each member of the container. In this case, the class is used as a kind of manually refreshed cache to avoid excessive recomputation. The other way of creating an instance of this class is through the use of the `ILOMULTIPLEEVALUATOR0` macro (or one of its variants), which allows complete freedom in how the evaluations are derived from the objects in the container class.

See Also: ILOMULTIPLEEVALUATOR0

Constructor Summary	
`public`	`IloMultipleEvaluator(IloEnv env, IloEvaluator< IloObject > evaluator, IloVisitor< IloObject, IloContainer > visitor=0)` Builds a multiple evaluator from a subordinate evaluator.

Method Summary	
`public void`	`update(IloContainer container) const` Updates the evaluations of the invoking evaluator.

Inherited Methods from `IloExplicitEvaluator`
`getEvaluation, getNumberOfEvaluations, hasEvaluation, removeAllEvaluations, removeEvaluation, setEvaluation`

Inherited Methods from `IloEvaluator`
`makeGreaterThanComparator, makeLessThanComparator, operator()`

Constructors

`public` **`IloMultipleEvaluator`**`(IloEnv env, IloEvaluator< IloObject > evaluator, IloVisitor< IloObject, IloContainer > visitor=0)`

Builds a multiple evaluator from a subordinate evaluator.

This constructor builds a multiple evaluator for an environment `env`, an evaluator `evaluator` and an optional visitor `visitor`. The resulting evaluator will evaluate its objects using `evaluator`. The visitor `visitor` will be used to traverse the containers used used to update the evaluator. If no visitor is specified, a default visitor will be used if it exists. If no default visitor exists, an exception of type `IloException` is raised.

Methods

`public void` **`update`**`(IloContainer container) const`

Updates the evaluations of the invoking evaluator.

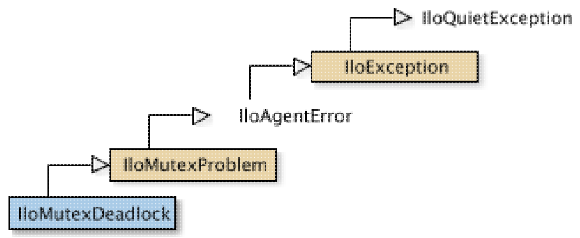
This member function updates the evaluations stored in the invoking evaluator for each member of the container `container`. The member function has two behaviors depending on how the class was created. However, common to those two behaviors is the initial removal of all evaluations currently contained in the invoking evaluator.

If the class was created using the `ILOMULTIPLEEVALUATOR0` macro (or one of its variants), the user-defined method of the macro is called, with `container` passed as argument. This user-defined method will use calls to `IloExplicitEvaluator::setEvaluation` to fill the evaluator.

If the class was created from an evaluator, then the visitor (or the default visitor if none was specified) will be used to traverse `container`. For each object visited, its evaluation will be performed by `evaluator` passed at construction time, and its evaluation will be stored in the invoking evaluator.

Class IloMutexDeadlock

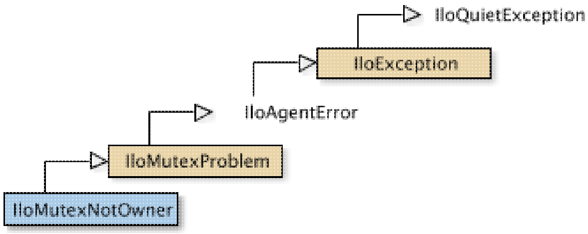
Definition file: ilconcert/ilothread.h



The class of exceptions thrown due to mutex deadlock.
This is the class of exceptions thrown if two or more threads become deadlocked waiting for a mutex owned by the other(s).

Class IloMutexNotOwner

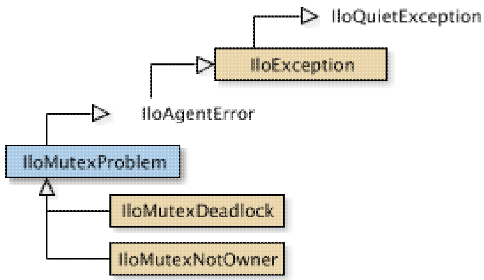
Definition file: ilconcert/ilothread.h



The class of exceptions thrown.
The class of exceptions thrown if a thread attempts to unlock a mutex that it does not own.

Class IloMutexProblem

Definition file: ilconcert/ilothread.h



Exception.
The class `IloMutexProblem` is part of the hierarchy of classes representing exceptions in Concert Technology. Concert Technology uses instances of this class when an error occurs with respect to a mutex, an instance of `IloFastMutex`.

An exception is thrown; it is not allocated in a Concert Technology environment; it is not allocated on the C++ heap. It is not necessary for you as a programmer to delete an exception explicitly. Instead, the system calls the constructor of the exception to create it, and the system calls the destructor of the exception to delete it.

When exceptions are enabled on a platform that supports C++ exceptions, an instance of `IloMutexProblem` makes it possible for Concert Technology to throw an exception in case of error. On platforms that do not support C++ exceptions, an instance of this class makes it possible for Concert Technology to exit in case of error.

Throwing and Catching Exceptions

Exceptions are thrown by value. They are not allocated on the C++ heap, nor in a Concert Technology environment. The correct way to catch an exception is to catch a reference to the error (specified by the ampersand `&`), like this:

```
catch(IloMutexProblem& error);
```

See Also: IloException, IloFastMutex

Constructor Summary	
public	IloMutexProblem(const char * msg)

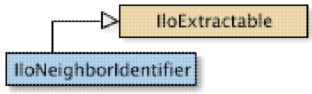
Constructors

`public IloMutexProblem(const char * msg)`

This constructor creates an instance of IloMutexProblem to represent an exception in case of an error involving a mutex. This instance is not allocated on C++ heap; it is not allocated in a Concert Technology environment either.

Class IloNeighborIdentifier

Definition file: ilsolver/iimnhood.h
Include file: <ilsolver/local.h>



This class communicates information between instances of local search goals. This class is used by `IloScanNHood` and `IloScanDeltas` to store information about the currently instantiated neighbor. This class is used by `IloTest` and `IloNotify` to find out the details of the currently instantiated neighbor. This information can be communicated by passing the same instance of `IloNeighborIdentifier` to such goals in a composed goal.

An instance of this class is extracted by an instance of `IloSolver` to an instance of `IlcNeighborIdentifier`.

See Also: IloScanNHood, IloScanDeltas, IloSolver, IloTest, IloNotify, IloSingleMove, IlcNeighborIdentifier, IloIIM

Constructor Summary	
`public`	`IloNeighborIdentifier()`
`public`	`IloNeighborIdentifier(IloNeighborIdentifierI * impl)`
`public`	`IloNeighborIdentifier(IloEnv env)`

Method Summary	
`public IloNeighborIdentifierI *`	`getImpl() const`

Inherited Methods from `IloExtractable`
`asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject`

Constructors

`public` **`IloNeighborIdentifier`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IloNeighborIdentifier`**`(IloNeighborIdentifierI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public` **`IloNeighborIdentifier`**`(IloEnv env)`

This constructor creates an instance of `IloNeighborIdentifier` associated with `env`.

Methods

`public IloNeighborIdentifierI *` **`getImpl`**`() const`

This member function returns a pointer to the implementation object of the invoking handle.

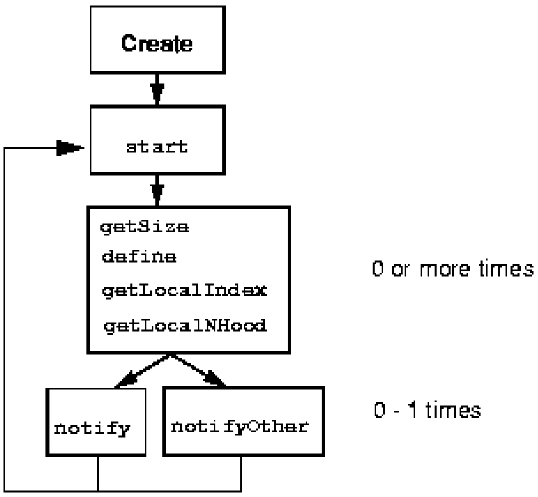
Class IloNHood

Definition file: ilsolver/iimnhood.h
Include file: <ilsolver/iimls.h>

This handle class is used to define neighborhood structures that can be used by local search procedures. The central idea of the neighborhood is to define a set of solution changes, or *deltas*, that represent alternative moves that can be taken.

Protocol

The protocol for driving an object of type `IloNHood` is as follows:



For more information, see `IloSolution` in the *Concert Technology Reference Manual*.

See Also: IloConcatenate, IloNHoodI, IloSample, IloScanDeltas, IloScanNHood

Constructor Summary	
public	`IloNHood()`
public	`IloNHood(IloNHoodI * impl)`
public	`IloNHood(const IloNHood & nhood)`

Method Summary	
public IloSolution	`define(IloSolver solver, IloInt index)`
public void	`end(IloBool deep=IloTrue)`
public IloEnv	`getEnv() const`
public IloNHoodI *	`getImpl() const`
public IloInt	`getLocalIndex(IloSolver solver, IloInt index) const`
public IloNHood	`getLocalNHood(IloSolver solver, IloInt index) const`
public const char *	`getName() const`
public IloAny	`getObject() const`

public IloInt	getSize(IloSolver solver) const
public void	notify(IloSolver solver, IloInt index) const
public void	notifyOther(IloSolver solver, IloSolution delta) const
public void	operator=(const IloNHood & nhood)
public void	reset()
public void	setName(const char * name) const
public void	setObject(IloAny obj) const
public void	start(IloSolver solver, IloSolution current) const

Constructors

public **IloNHood**()

This constructor creates a neighborhood whose implementation pointer is 0 (zero). The handle must be assigned before its methods can be used.

public **IloNHood**(IloNHoodI * impl)

This constructor creates a handle object (an instance of the class `IloNHood`) from a pointer to an implementation object (an instance of the class `IloNHoodI`).

public **IloNHood**(const IloNHood & nhood)

This constructor creates a handle object from a reference to a neighborhood. After execution, both the newly constructed handle object and `nhood` point to the same implementation object.

Methods

public IloSolution **define**(IloSolver solver, IloInt index)

This member function returns the change in the current solution to be made when making move `index` in the neighborhood. The neighbors are numbered from 0. If an empty handle is returned, no change is defined for this index.

The argument solver indicates the solver that is scanning the invoking neighborhood. (See `IloScanNHood`.)

public void **end**(IloBool deep=IloTrue)

This member function calls `delete` on the implementation object associated with the invoking handle and sets the implementation pointer to 0. If the invoking neighborhood has any children (for instance, if it were constructed using a + operator) then these too will be likewise deleted. If, however, `deep` has the value `IloFalse`, then the children are “disconnected” from the parent before `delete` is called on the implementation pointer to ensure that the children of the invoking neighborhood are not destroyed.

public IloEnv **getEnv**() const

This member function returns the environment associated with the implementation class.

`public IloNHoodI * getImpl() const`

This member function returns the implementation object of the invoking handle. You can use this member function to check whether a neighborhood is empty.

`public IloInt getLocalIndex(IloSolver solver, IloInt index) const`

This member function converts the index `index` to the index in the atomic neighborhood responsible for defining neighbor `index`. This member function is useful for finding the index in the neighborhood that defined a neighbor when the invoking neighborhood is made up of neighborhoods combined via concatenation (operator `operator+` or `IloConcatenate`). It is also useful when the invoking neighborhood has been modified (for instance via `IloRandomize`). In typical usage, the `index` passed originates from an instance of `IloNeighborIdentifier`.

`public IloNHood getLocalNHood(IloSolver solver, IloInt index) const`

This member function returns the atomic neighborhood responsible for defining the neighbor indexed `index`. This member function is useful for finding the exact neighborhood that defined a neighbor when the invoking neighborhood is made up of neighborhoods combined via (`operator+` or `IloConcatenate`). In typical usage, the `index` passed originates from an instance of `IloNeighborIdentifier`.

`public const char * getName() const`

This member function returns a character string specifying the name of the invoking object (if there is one).

`public IloAny getObject() const`

This member function returns the object associated with the invoking object (if there is one). Normally, an associated object contains user data pertinent to the invoking object.

`public IloInt getSize(IloSolver solver) const`

This member function returns the "size" of the neighborhood, that is, one more than the maximum index that `IloNHood::define` can legally be called with. The argument solver indicates the solver that is scanning the invoking neighborhood. (See `IloScanNHood`.)

`public void notify(IloSolver solver, IloInt index) const`

This member function is called when a neighborhood move is about to be taken. It is called with `index` equal to the index of the neighbor currently being accepted by local search. The argument solver indicates the solver that is scanning the invoking neighborhood. (See `IloScanNHood`.)

At the point of calling, the constrained variables corresponding to the solution passed to `IloNHood::start` should be instantiated, but not yet been saved to the current solution. This allows inspection of the differences. Default behavior is to do nothing.

`public void notifyOther(IloSolver solver, IloSolution delta) const`

This member function is a counterpart to `IloNHood::notify`. When neighborhoods are joined with `operator+` or `IloConcatenate`, only one basic neighborhood can have `IloNHood::notify` called: the one that defined the delta for the move to be taken. For all other basic neighborhoods, `notifyOther` is called with the delta for the move to be taken.

The argument `solver` indicates the solver that is scanning the invoking neighborhood. (See `IloScanNHood`.)

`public void operator=(const IloNHood & nhood)`

This assignment operator copies `nhood` into the invoking neighborhood by assigning an address to the handle pointer of the invoking object. That address is the location of the implementation object of the argument `nhood`.

`public void reset()`

This member function resets the neighborhood to the state in which it was first created. For neighborhoods which have no state, calling this member function has no effect. An example of a neighborhood with state is `IloContinue` which, after each move, adjusts neighborhood indices to ensure that new indices are explored. When neighborhoods have state, you will typically want to call this member function between two different local searches if the neighborhood is to be reused. Most of Solver's neighborhoods are stateless.

`public void setName(const char * name) const`

This member function assigns `name` to the invoking object.

`public void setObject(IloAny obj) const`

This member function associates `obj` with the invoking object. The member function `getObject` accesses this associated object afterward. Normally, `obj` contains user data pertinent to the invoking object.

`public void start(IloSolver solver, IloSolution current) const`

This member function announces to the neighborhood that neighbors will be requested of it (through `IloNHood::define`), and that `current` is the reference point of these changes. That is, all changes specified afterwards via calls to `IloNHood::define` (until `start` is called again) should relate to changes of `current`.

The argument solver indicates the solver that is scanning the invoking neighborhood. (See `IloScanNHood`.)

Class IloNHoodArray

Definition file: ilsolver/iimnhood.h
Include file: <ilsolver/iimls.h>

The class `IloNHoodArray` is the class for an array of instances of `IloNHood`.

See Also: IloNHood, IloConcatenate, IloArray

Constructor Summary	
`public`	`IloNHoodArray()`
`public`	`IloNHoodArray(IloNHoodArrayI * impl)`
`public`	`IloNHoodArray(const IloNHoodArray & array)`
`public`	`IloNHoodArray(IloEnv env, IloInt size)`

Method Summary	
`public IloEnv`	`getEnv() const`
`public IloNHoodArrayI *`	`getImpl() const`
`public IloInt`	`getSize() const`
`public void`	`operator=(const IloNHoodArray & array)`
`public IloNHood`	`operator[](IloInt i) const`
`public IloNHood &`	`operator[](IloInt i)`

Constructors

`public` **`IloNHoodArray`**`()`

This constructor creates an uninitialized array of neighborhoods. The index range of the array is undefined. The array must be assigned before it can be used.

`public` **`IloNHoodArray`**`(IloNHoodArrayI * impl)`

This constructor creates a handle object (an instance of the class `IloNHoodArray`) from a pointer to an object (an instance of the implementation class `IloNHoodArrayI`).

`public` **`IloNHoodArray`**`(const IloNHoodArray & array)`

This copy constructor creates a handle from a reference to a neighborhood array. That neighborhood array and `array` both point to the same implementation object.

`public` **`IloNHoodArray`**`(IloEnv env, IloInt size)`

This constructor creates an array of `size` neighborhoods associated with environment `env`. The index range of the array is `[0..size)`, where 0 is included but `size` is excluded. The argument `size` must be strictly greater than 0 (zero).

Methods

`public IloEnv **getEnv**() const`

This member function returns the environment associated with the invoking array.

`public IloNHoodArrayI * **getImpl**() const`

This member function returns the implementation object of the invoking handle. You can use this member function to check whether an array is empty.

`public IloInt **getSize**() const`

This member function returns the number of neighborhoods in the invoking array.

`public void **operator=**(const IloNHoodArray & array)`

This assignment operator copies `array` into the invoking array by assigning an address to the handle pointer of the invoking object. That address is the location of the implementation object of the argument `array`.

`public IloNHood **operator[]**(IloInt i) const`

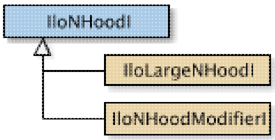
This subscripting operator returns the neighborhood corresponding to rank `i` in the invoking neighborhood array.

`public IloNHood & **operator[]**(IloInt i)`

This operator returns a reference to the element at rank `i`. This operator can be used for accessing (that is, simply reading) the element or for modifying (that is, writing) it.

Class IloNHoodI

Definition file: ilsolver/iimnhood.h
Include file: <ilsolver/iimls.h>



This abstract implementation class is used to describe a neighborhood structure that can be used by local search procedures. The central idea of the neighborhood is to define a set of solution changes, or *deltas*, that represent alternative moves that can be taken.

You can subclass the `IloNHoodI` class to define your own neighborhoods.

The member functions `IloNHoodI::start`, `IloNHoodI::define`, `IloNHoodI::getSize`, `IloNHoodI::notify`, and `IloNHoodI::notifyOther` are called within search, and the solver calling these is passed as the first parameter of these member functions. Solver automatically deletes any deltas you return from the `define` method. Therefore, you do not need to handle the memory management of these deltas. However, this also means that you should not make any further reference to a delta that has been returned from the `define`.

For more information, see `IloSolution` in the *Concert Technology Reference Manual*.

See Also: IloNHood, IloScanDeltas, IloScanNHood

Constructor and Destructor Summary	
`public`	`IloNHoodI(IloEnv env, const char * name=0)`
`public`	`~IloNHoodI()`

Method Summary	
`public virtual IloSolution`	`define(IloSolver solver, IloInt index)`
`public virtual void`	`display(ostream & stream) const`
`public IloEnv`	`getEnv() const`
`public virtual IloInt`	`getLocalIndex(IloSolver solver, IloInt index) const`
`public virtual IloNHoodI *`	`getLocalNHood(IloSolver solver, IloInt index) const`
`public char *`	`getName() const`
`public IloAny`	`getObject()`
`public virtual IloInt`	`getSize(IloSolver solver) const`
`public virtual void`	`notify(IloSolver solver, IloInt index)`
`public virtual void`	`notifyOther(IloSolver solver, IloSolution solution)`
`public void`	`operator delete(void * p, size_t size)`
`public virtual void`	`reset()`
`public void`	`setName(const char * name)`
`public void`	`setObject(IloAny object)`
`public virtual void`	`start(IloSolver solver, IloSolution currentSolution)`

Constructors and Destructors

public **IloNHoodI**(IloEnv env, const char * name=0)

This constructor creates a neighborhood. The optional argument `name`, if supplied, becomes the name of the neighborhood.

public **~IloNHoodI**()

Since `IloNHoodI` is an abstract class, a virtual destructor is provided.

Methods

public virtual IloSolution **define**(IloSolver solver, IloInt index)

This virtual member function should return the change in the current solution to be made when making move number `index` in the neighborhood. The neighbors are numbered from 0.

public virtual void **display**(ostream & stream) const

This virtual member function displays the name of the neighborhood, if named. Otherwise, it prints "`IloNHoodI`", followed by the address of the invoking object in brackets.

public IloEnv **getEnv**() const

This member function returns the environment which was passed in the constructor.

public virtual IloInt **getLocalIndex**(IloSolver solver, IloInt index) const

This member function should convert the index `index` to the index in the atomic neighborhood responsible for defining neighbor `index`. If you define a neighborhood that is a combination of other neighborhoods, you should define this member function to produce the correct index. By default, this member function returns `index`.

public virtual IloNHoodI * **getLocalNHood**(IloSolver solver, IloInt index) const

This member function should return the atomic neighborhood responsible for defining the neighbor indexed `index`. If you define a neighborhood that is a combination of other neighborhoods, you should define this member function to produce the correct atomic neighborhood. By default, this member function returns the invoking neighborhood.

public char * **getName**() const

This member function returns the name specified in the constructor of the object, or specified in the last call to `setName`.

`public IloAny getObject()`

This member function returns the object associated with the invoking neighborhood through the `IloNHoodI::setObject` method.

`public virtual IloInt getSize(IloSolver solver) const`

This pure virtual member function should return the “size” of the neighborhood, that is, one more than the maximum index that `define` can be legally called with.

`public virtual void notify(IloSolver solver, IloInt index)`

This member function is called when a neighborhood move is about to be taken. It is called with `index` equal to the index of the neighbor currently being accepted by local search. If you are using `IloSingleMove`, then at the point of calling the constrained variables are instantiated, but not yet saved in the current solution. This permits inspection of differences between the current solution and the one about to be accepted. The default behavior of this member function is to do nothing.

`public virtual void notifyOther(IloSolver solver, IloSolution solution)`

This member function is a counterpart to `IloNHoodI::notify`. When neighborhoods are joined with `operator+` or `IloConcatenate`, only one basic neighborhood can have `IloNHoodI::notify` called: the one that defined the delta for the move to be taken. For all other basic neighborhoods, `notifyOther` is called with the delta for the move to be taken.

The argument solver indicates the solver that is scanning the invoking neighborhood. (See `IloScanNHood`.)

`public void operator delete(void * p, size_t size)`

This operator deletes the memory for the neighborhood pointed to by `p` as if its memory was allocated using the environment handed to the neighborhood when it was created.

If you want to allocate your neighborhoods on a different heap when subclassing, you should redefine this operator in the subclass to perform the delete operation appropriate to your heap.

`public virtual void reset()`

If your neighborhood has state which it carries from move to move, you should define this member function to reset the neighborhood to the state in which it was first created.

`public void setName(const char * name)`

This member function sets the name of the invoking object to a copy of `name`.

`public void setObject(IloAny object)`

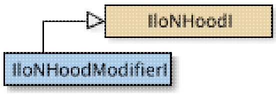
This member function associates object `object` with the invoking neighborhood.

`public virtual void start(IloSolver solver, IloSolution currentSolution)`

This pure virtual member function announces to the neighborhood that neighbors that neighbors will be requested of it (through `IloNHoodI::define`) and that `currentSolution` is the reference point of these changes. That is, all changes specified afterwards via calls `IloNHoodI::define`) (until `start` is called again) should related to changes of `currentSolution`.

Class IloNHoodModifierI

Definition file: ilsolver/iimnhood.h
Include file: <ilsolver/iimls.h>



This abstract implementation class is used to describe a neighborhood structure that depends on other “child” neighborhood structures. Solver neighborhoods such as `IloConcatenate`, `IloRandomize`, and `IloContinue` belong to this category of neighborhood. Such neighborhoods do not generate neighbors in their own right, but pass back neighbors produced by their children. The main job of a neighborhood modifier is to maintain a mapping between the indices of neighbors for the neighborhood modifier and the corresponding child neighborhoods and indices within these child neighborhoods. The member function IloNHoodModifierI::mapIndex performs this maintenance. Additional behaviors can be added. One example is to manipulate normally produced deltas in some way, for instance by adding additional variables, before they are returned.

See Also: IloNHood, IloNHoodI

Constructor and Destructor Summary	
public	`IloNHoodModifierI(IloEnv env, IloNHoodArray a, const char * name=0)`
public	`IloNHoodModifierI(IloEnv env, IloNHood nh, const char * name=0)`
public	`IloNHoodModifierI(IloEnv env, IloNHood nh1, IloNHood nh2, const char * name=0)`

Method Summary	
public IloSolution	`define(IloSolver solver, IloInt index)`
public void	`display(ostream & out) const`
public IloInt	`getLocalIndex(IloSolver solver, IloInt index) const`
public IloNHoodI *	`getLocalNHood(IloSolver solver, IloInt index) const`
public IloNHood	`getNHood(IloInt i) const`
public IloInt	`getNHoodSize(IloInt i) const`
public IloInt	`getNumberOfNHoods() const`
public IloInt	`getSize(IloSolver solver) const`
public virtual void	`mapIndex(IloSolver solver, IloInt index, IloNHood & nhood, IloInt & offset) const`
public void	`notify(IloSolver solver, IloInt index)`
public void	`notifyOther(IloSolver solver, IloSolution delta)`
public void	`reset()`
public void	`start(IloSolver solver, IloSolution solution)`

Inherited Methods from `IloNHoodI`
`define, display, getEnv, getLocalIndex, getLocalNHood, getName, getObject, getSize, notify, notifyOther, operator delete, reset, setName, setObject, start`

Constructors and Destructors

`public **IloNHoodModifierI**(IloEnv env, IloNHoodArray a, const char * name=0)`

This constructor builds a neighborhood modifier that combines all neighborhoods in the array `a`. The optional argument `name`, if provided, becomes the name of the neighborhood.

`public **IloNHoodModifierI**(IloEnv env, IloNHood nh, const char * name=0)`

This constructor builds a neighborhood modifier that depends on just one neighborhood `nh`. The optional argument `name`, if provided, becomes the name of the neighborhood.

`public **IloNHoodModifierI**(IloEnv env, IloNHood nh1, IloNHood nh2, const char * name=0)`

This constructor builds a neighborhood modifier that depends on two neighborhoods, `nh1` and `nh2`. The optional argument `name`, if provided, becomes the name of the neighborhood.

Methods

`public IloSolution **define**(IloSolver solver, IloInt index)`

This member function calls `IloNHoodModifierI::define` on one of its child neighborhoods with an appropriately adjusted index. The computation of the child neighborhood and adjusted index are performed by the IloNHoodModifierI::mapIndex member function. If you wish `define` to have some additional behavior, you can redefine this function. In this case, you must still call `IloNHoodModifierI::define(solver, index)` in your subclass and return the delta delivered (or a modified version of it).

`public void **display**(ostream & out) const`

This virtual member function displays the name of the neighborhood, if named. Otherwise, it prints "`IloNHoodI`", followed by the address of the invoking object in brackets.

`public IloInt **getLocalIndex**(IloSolver solver, IloInt index) const`

This member function converts the index `index` to the index in the atomic neighborhood responsible for defining neighbor `index`. To do this, it uses the member function `IloNHoodModifierI::mapIndex`.

`public IloNHoodI * **getLocalNHood**(IloSolver solver, IloInt index) const`

This member function returns the atomic neighborhood responsible for defining the neighbor indexed `index`. To do this, it uses the member function `IloNHoodModifierI::mapIndex`.

`public IloNHood **getNHood**(IloInt i) const`

This member function delivers the *ith* neighborhood specified at construction time. If you constructed the neighborhood modifier with an array, this member function returns the *ith* element of that array. If you constructed the neighborhood modifier with two neighborhoods, `getNHood(0)` returns `nhood0` and `getNHood(1)` returns `nhood1`. Finally, if you constructed the neighborhood modifier with only one neighborhood, `getNHood(0)` returns that neighborhood.

`public IloInt` **`getNHoodSize`**`(IloInt i) const`

This member function returns the size of the *ith* neighborhood specified at construction time. The indexing rules for neighborhoods are the same as those described in the description for the member function `getNHood`. Although this member function is functionally equivalent to `getNHood(i).getSize(solver)` (given an instance of `IloSolversolver`), it can be faster as the size is cached in the neighborhood modifier and not recomputed from the child neighborhood.

`public IloInt` **`getNumberOfNHoods`**`() const`

This member function returns the number of neighborhoods specified at construction time. The value returned is one greater than the legal maximum value with which you can call `IloNHoodModifierI::getNHood` or `IloNHoodModifierI::getNHoodSize`.

`public IloInt` **`getSize`**`(IloSolver solver) const`

This member function returns the sum of the sizes of the child neighborhoods. You may redefine this behavior if desired.

`public virtual void` **`mapIndex`**`(IloSolver solver, IloInt index, IloNHood & nhood, IloInt & offset) const`

This member function maintains a mapping between the indices of neighbors for the neighborhood modifier and the corresponding child neighborhoods and indices within these child neighborhoods. You should thus redefine this member function to convert the index `index` to a neighborhood `nhood` and an index `offset` within that neighborhood. `nhood` and `offset` are passed by reference so that you can fill in the details. `solver` is the active solver. An example of how to do this for a modifier which concatenates two neighborhoods is shown in the *Solver User's Manual*.

`public void` **`notify`**`(IloSolver solver, IloInt index)`

This member function calls `IloNHoodModifierI::notify` on one of its child neighborhoods with an appropriately adjusted index. The computation of the child neighborhood and adjusted index are performed by the `IloNHoodModifierI::mapIndex` member function. For all other child neighborhoods, `IloNHoodModifierI::notifyOther` is called. If you wish `notify` to have some additional behavior, you can redefine this function. In this case, you must still call `IloNHoodModifierI::notify(solver, index)` in your subclass.

`public void` **`notifyOther`**`(IloSolver solver, IloSolution delta)`

This member function calls `notifyOther` on all child neighborhoods. If you wish `notifyOther` to have some additional behavior, you can redefine this function. In this case, you must still call `IloNHoodModifierI::notifyOther(solver, delta)` in your subclass.

`public void **reset**()`

This member function calls `IloNHoodModifierI::reset` on all child neighborhoods.

`public void **start**(IloSolver solver, IloSolution solution)`

This member function calls `start` on all child neighborhoods passing the parameters it receives. It then calls `getSize` on all child neighborhoods and caches the values returned. These sizes can be later retrieved by `getNHoodSize`. If you wish `start` to have some additional behavior, you can redefine this function. In this case, you must still call `IloNHoodModifierI::start(solver, solution)` in your subclass.

Class IloNodeEvaluator

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

An instance of this class represents a *node evaluator* in a Concert Technology model. A node evaluator offers a way for an algorithm to discriminate among nodes of the search tree during the search for a solution. A node evaluator is useful in functions such as `IloApply`.

There are predefined functions in IBM® ILOG® Solver that create and return a node evaluator: `IloBFSEvaluator`, `IloDDSEvaluator`, `IloDFSEvaluator`, `IloIDFSEvaluator`, and `IloSBSEvaluator`.

See Also: IloApply, IloBFSEvaluator, IloDDSEvaluator, IloDFSEvaluator, IloSBSEvaluator

Constructor Summary	
public	IloNodeEvaluator()
public	IloNodeEvaluator(IloNodeEvaluatorI * impl)

Method Summary	
public void	end() const
public IloNodeEvaluatorI *	getImpl() const
public void	operator=(const IloNodeEvaluator & h)

Constructors

`public` **`IloNodeEvaluator`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IloNodeEvaluator`**`(IloNodeEvaluatorI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

Methods

`public void` **`end`**`() const`

This member function ends the corresponding node evaluator and returns the memory to the environment.

`public IloNodeEvaluatorI *` **`getImpl`**`() const`

This constructor creates an object by copying another one.

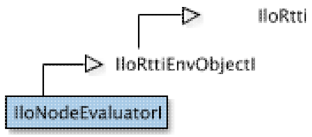
This member function returns a pointer to the implementation object of the invoking handle.

`public void` **`operator=`**`(const IloNodeEvaluator & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IloNodeEvaluatorI

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>



The class `IloNodeEvaluator` represents objects to evaluate whether or not to explore a node during a Solver search in an IBM® ILOG® Concert Technology *model*. The class `IlcNodeEvaluator` represents node evaluators internally in a Solver search.

A node evaluator is an object in Solver. Like other Solver entities, a node evaluator is implemented by means of two classes: a handle class and an implementation class. In other words, an instance of the class `IloNodeEvaluator` (a handle) contains a data member (the handle pointer) that points to an instance of the class `IloNodeEvaluatorI` (its implementation object).

See Also: IloNodeEvaluator

Constructor and Destructor Summary	
`public`	`~IloNodeEvaluatorI()`

Method Summary	
`public virtual void`	`display(ostream &) const`
`public virtual IlcNodeEvaluator`	`extract(const IloSolver solver) const`
`public virtual IloNodeEvaluatorI *`	`makeClone(IloEnvI * env) const`

Constructors and Destructors

`public` **`~IloNodeEvaluatorI`**`()`

This destructor is called automatically by the destructor of its subclasses. It frees memory used by the invoking object.

Methods

`public virtual void` **`display`**`(ostream &) const`

This member function prints the invoking node evaluator on an output stream.

`public virtual IlcNodeEvaluator` **`extract`**`(const IloSolver solver) const`

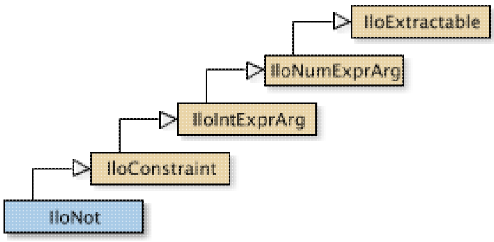
In general terms, in Concert Technology, the objects of a model must be extracted for an algorithm (an instance of one of the subclasses of `IloAlgorithm`, such as `IloSolver`). This member function returns the internal node evaluator extracted for `solver` from the invoking node evaluator of a model.

`public virtual IloNodeEvaluatorI *` **`makeClone`**`(IloEnvI * env) const`

This member function is called internally to duplicate the current node evaluator.

Class IloNot

Definition file: ilconcert/ilomodel.h



Negation of its argument.
The class `IloNot` represents a constraint that is the negation of its argument. In order to be taken into account, this constraint must be added to a model and extracted by an algorithm, such as `IloCplex` or `IloSolver`.

See Also: operator!

Constructor Summary	
`public`	`IloNot()`
`public`	`IloNot(IloNotI * impl)`

Method Summary	
`public IloNotI *`	`getImpl() const`

Inherited Methods from `IloConstraint`
`getImpl`

Inherited Methods from `IloIntExprArg`
`getImpl`

Inherited Methods from `IloNumExprArg`
`getImpl`

Inherited Methods from `IloExtractable`
`asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject`

Constructors

`public` **`IloNot`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IloNot`**`(IloNotI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

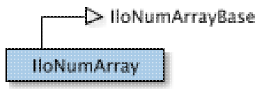
Methods

```
public IloNotI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

Class IloNumArray

Definition file: ilconcert/iloenv.h



The array class of the basic floating-point class.
For each basic type, Concert Technology defines a corresponding array class. `IloNumArray` is the array class of the basic floating-point class (`IloNum`) for a model.

Instances of `IloNumArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

`IloNumArray` inherits additional methods from the template `IloArray`:

- `IloArray::add`
- `IloArray::add`
- `IloArray::add`
- `IloArray::clear`
- `IloArray::getEnv`
- `IloArray::getSize`
- `IloArray::remove`
- `IloArray::operator[]`
- `IloArray::operator[]`

See Also: IloNum, operator>>, operator<<

Constructor Summary	
`public`	`IloNumArray(IloArrayI * i=0)`
`public`	`IloNumArray(const IloNumArray & cpy)`
`public`	`IloNumArray(const IloEnv env, IloInt n=0)`
`public`	`IloNumArray(const IloEnv env, IloInt n, IloNum f0, IloNum f1, ...)`

Method Summary	
`public IloBool`	`contains(IloNum value) const`
`public IloNum &`	`operator[](IloInt i)`
`public const IloNum &`	`operator[](IloInt i) const`
`public IloNumExprArg`	`operator[](IloIntExprArg intExp) const`
`public IloIntArray`	`toIntArray() const`

Constructors

`public` **`IloNumArray`**`(IloArrayI * i=0)`

This constructor creates an empty array of floating-point numbers for use in a model. You cannot create instances of the undocumented class `IloDefaultArrayI`. As an argument in this default constructor, it allows you to pass 0 (zero) as a value to an optional argument in functions and member functions that accept an array as an argument.

`public IloNumArray(const IloNumArray & cpy)`

This copy constructor creates a handle to the array of floating-point objects specified by `cpy`.

`public IloNumArray(const IloEnv env, IloInt n=0)`

This constructor creates an array of `n` elements. Initially, the `n` elements are empty handles.

`public IloNumArray(const IloEnv env, IloInt n, IloNum f0, IloNum f1, ...)`

This constructor creates an array of `n` floating-point objects for use in a model.

Methods

`public IloBool contains(IloNum value) const`

This member function checks whether the value is contained or not.

`public IloNum & operator[](IloInt i)`

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`.

`public const IloNum & operator[](IloInt i) const`

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`. On `const` arrays, Concert Technology uses the `const` operator:

```
IloArray operator[] (IloInt i) const;
```

`public IloNumExprArg operator[](IloIntExprArg intExp) const`

This subscripting operator returns an expression node for use in a constraint or expression. For clarity, let's call `A` the invoking array. When `intExp` is bound to the value `i`, then the domain of the expression is the domain of `A[i]`. More generally, the domain of the expression is the union of the domains of the expressions `A[i]` where the `i` are in the domain of `intExp`.

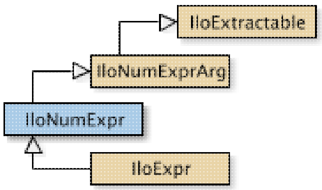
This operator is also known as an element expression.

`public IloIntArray toIntArray() const`

This member function copies the invoking numeric array to a new instance of `IloIntArray`, checking the type of the values during the copy.

Class IloNumExpr

Definition file: ilconcert/iloexpression.h



The class of numeric expressions in a Concert model.
Numeric expressions in Concert Technology are represented using the class `IloNumExpr`.

Constructor Summary	
`public`	`IloNumExpr()`
`public`	`IloNumExpr(IloNumExprI * impl)`
`public`	`IloNumExpr(const IloNumExprArg expr)`
`public`	`IloNumExpr(const IloEnv env, IloNum=0)`
`public`	`IloNumExpr(const IloNumLinExprTerm term)`
`public`	`IloNumExpr(const IloIntLinExprTerm term)`
`public`	`IloNumExpr(const IloExpr & expr)`

Method Summary	
`public IloNumExprI *`	`getImpl() const`
`public IloNumExpr &`	`operator*=(IloNum val)`
`public IloNumExpr &`	`operator+=(const IloNumExprArg expr)`
`public IloNumExpr &`	`operator+=(IloNum val)`
`public IloNumExpr &`	`operator-=(const IloNumExprArg expr)`
`public IloNumExpr &`	`operator-=(IloNum val)`
`public IloNumExpr &`	`operator/=(IloNum val)`

Inherited Methods from `IloNumExprArg`
`getImpl`

Inherited Methods from `IloExtractable`
`asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject`

Inner Class	
IloNumExpr::NonLinearExpression	The class of exceptions thrown if a numeric constant of a nonlinear expression is set or queried.

Constructors

`public IloNumExpr()`

This constructor creates an empty handle. You must initialize it before you use it.

`public IloNumExpr(IloNumExprI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public IloNumExpr(const IloNumExprArg expr)`

This constructor creates a numeric expression using the undocumented class `IloNumExprArg`.

`public IloNumExpr(const IloEnv env, IloNum=0)`

This constructor creates a constant numeric expression with the value `n` that the user can modify subsequently with the operators `+=, -=, =` in the environment specified by `env`. It may be used to build other expressions from variables belonging to `env`.

`public IloNumExpr(const IloNumLinExprTerm term)`

This constructor creates a numeric expression using the undocumented class `IloNumLinExprTerm`.

`public IloNumExpr(const IloIntLinExprTerm term)`

This constructor creates a numeric expression using the undocumented class `IloIntLinExprTerm`.

`public IloNumExpr(const IloExpr & expr)`

This is the copy constructor for this class.

Methods

`public IloNumExprI * getImpl() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public IloNumExpr & operator*=(IloNum val)`

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x * ...`

`public IloNumExpr & operator+=(const IloNumExprArg expr)`

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x + ...`

`public IloNumExpr & operator+=(IloNum val)`

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x + ...`

`public IloNumExpr & operator-=(const IloNumExprArg expr)`

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x - ...`

`public IloNumExpr & operator-=(IloNum val)`

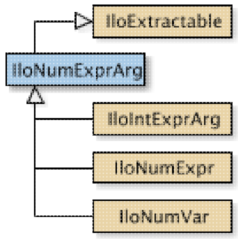
This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x - ...`

`public IloNumExpr & operator/=(IloNum val)`

This operator is recommended for building a Concert Technology expression in a loop. It is more efficient than `x = x / ...`

Class IloNumExprArg

Definition file: ilconcert/iloexpression.h



A class used internally in Concert Technology.
Concert Technology uses instances of this class internally as temporary objects when it is parsing a C++ expression in order to build an instance of `IloNumExpr`. As a Concert Technology user, you will not need this class yourself; in fact, you should not use them directly. They are documented here because the return value of certain functions, such as `IloSum` or `IloScalProd`, can be an instance of this class.

Constructor Summary	
public	IloNumExprArg()
public	IloNumExprArg(IloNumExprI * impl)

Method Summary	
public IloNumExprI *	getImpl() const

Inherited Methods from `IloExtractable`
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Constructors

public **IloNumExprArg**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IloNumExprArg**(IloNumExprI * impl)

This constructor creates a handle object from a pointer to an implementation object.

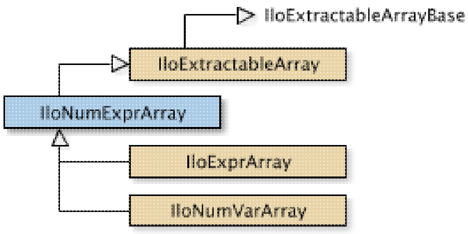
Methods

public IloNumExprI * **getImpl**() const

This member function returns a pointer to the implementation object of the invoking handle.

Class IloNumExprArray

Definition file: ilconcert/iloexpression.h



The array class of the numeric expressions class.
For each basic type, Concert Technology defines a corresponding array class. `IloNumExprArray` is the array class of the numeric expressions class (`IloNumExpr`) for a model.

Instances of `IloNumExprArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Constructor Summary	
`public`	`IloNumExprArray(IloDefaultArrayI * i=0)`
`public`	`IloNumExprArray(const IloEnv env, IloInt n=0)`

Method Summary	
`public void`	`add(IloInt more, const IloNumExpr x)`
`public void`	`add(const IloNumExpr x)`
`public void`	`add(const IloNumExprArray array)`
`public void`	`endElements()`
`public IloNumExprArg`	`operator[](IloIntExprArg anIntegerExpr) const`

Inherited Methods from `IloExtractableArray`
`add, add, add, endElements, setNames`

Constructors

`public` **`IloNumExprArray`**`(IloDefaultArrayI * i=0)`

This constructor creates an empty array of numeric expressions for use in a model. You cannot create instances of the undocumented class `IloDefaultArrayI`. As an argument in this default constructor, it allows you to pass 0 (zero) as a value to an optional argument in functions and member functions that accept an array as an argument.

`public` **`IloNumExprArray`**`(const IloEnv env, IloInt n=0)`

This constructor creates an array of `n` elements. Initially, the `n` elements are empty handles.

Methods

`public void add(IloInt more, const IloNumExpr x)`

This member function appends `x` to the invoking array. The argument `more` specifies how many times.

`public void add(const IloNumExpr x)`

This member function appends `x` to the invoking array.

`public void add(const IloNumExprArray array)`

This member function appends the elements in `array` to the invoking array.

`public void endElements()`

This member function calls `IloExtractable::end` for each of the elements in the invoking array. This deletes all the extractables identified by the array, leaving the handles in the array intact. This member function is the recommended way to delete the elements of an array.

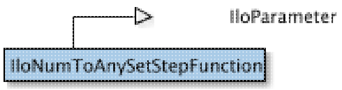
`public IloNumExprArg operator[](IloIntExprArg anIntegerExpr) const`

This subscripting operator returns an expression argument for use in a constraint or expression. For clarity, let's call `A` the invoking array. When `anIntegerExpr` is bound to the value `i`, the domain of the expression is the domain of `A[i]`. More generally, the domain of the expression is the union of the domains of the expressions `A[i]` where the `i` are in the domain of `anIntegerExpr`.

This operator is also known as an element expression.

Class IloNumToAnySetStepFunction

Definition file: ilconcert/ilosetfunc.h



Represents a step function that associates sets with intervals.
An instance of `IloNumToAnySetStepFunction` represents a step function that associates sets with intervals. It is defined everywhere on an interval *[xMin,xMax)*. Each interval *[x1,x2)* on which the function has the same set is called a step.

Note that if *n* is the number of steps of the function, the random access to a given step (see the member functions `IloNumToAnySetStepFunction::add`, `IloNumToAnySetStepFunction::alwaysIntersects`, `IloNumToAnySetStepFunction::contains`, `IloNumToAnySetStepFunction::empty`, `IloNumToAnySetStepFunction::everContains`, `IloNumToAnySetStepFunction::everIntersects`, `IloNumToAnySetStepFunction::fill`, `IloNumToAnySetStepFunction::getComplementSet`, `IloNumToAnySetStepFunction::getSet`, `IloNumToAnySetStepFunction::intersects`, `IloNumToAnySetStepFunction::isEmpty`, `IloNumToAnySetStepFunction::isFull`, `IloNumToAnySetStepFunction::remove`, `IloNumToAnySetStepFunction::set`, and `IloNumToAnySetStepFunction::setIntersection`) has a worst-case complexity of *O(log(n))*.

Complementary Representation of Values

`IloNumToAnySetStepFunction` allows the implicit representation of infinite sets through the representation of the complement of the actual set value. This, for example, allows you to completely fill a set (using the `IloNumToAnySetStepFunction::fill` member function) and then specify the elements that are not in the set. Under normal circumstances, it is not necessary to know if the value of the step function at a particular point is represented by the set or its complement: all the member functions that manipulate the step function value will correctly adapt to either representation. The only case where it is necessary to know the internal representation is if you want to directly access the set that represents a value (using the `IloNumToAnySetStepFunction::getSet` or `IloNumToAnySetStepFunction::getComplementSet` member functions). In that circumstance only, it is necessary to use the `IloNumToAnySetStepFunction::usesComplementaryRepresentation` member function to determine the internal representation, and then use either `IloNumToAnySetStepFunction::getSet` or `IloNumToAnySetStepFunction::getComplementSet` depending on the return value of `IloNumToAnySetStepFunction::usesComplementaryRepresentation`. Note that `IloNumToAnySetStepFunction::getSet` will raise an error if it is used to access a set that is represented as a complement set. `IloNumToAnySetStepFunction::getComplementSet` will raise an error if it is used to access a set that is directly represented.

See Also: IloNumToAnySetStepFunctionCursor

Constructor Summary	
public	`IloNumToAnySetStepFunction(const IloEnv env, IloNum xmin=-IloInfinity, IloNum xmax=IloInfinity, const char * name=0)`
public	`IloNumToAnySetStepFunction(const IloEnv env, IloNum xmin, IloNum xmax, const IloAnySet dval, const char * name=0)`

Method Summary	
public void	`add(const IloNumToAnySetStepFunction f) const`
public void	`add(IloNum xMin, IloNum xMax, IloAny elt) const`
public void	`add(IloNum xMin, IloNum xMax, const IloAnySet elts, IloBool complt=IloFalse) const`

public IloBool	alwaysContains(const IloNumToAnySetStepFunction f) const
public IloBool	alwaysContains(IloNum xMin, IloNum xMax, const IloAnySet elts) const
public IloBool	alwaysContains(IloNum xMin, IloNum xMax, IloAny elt) const
public IloBool	alwaysIntersects(const IloNumToAnySetStepFunction f) const
public IloBool	alwaysIntersects(IloNum xMin, IloNum xMax, const IloAnySet elts) const
public IloBool	contains(IloNum x, const IloAnySet elts) const
public IloBool	contains(IloNum x, IloAny elt) const
public IloNumToAnySetStepFunction	copy() const
public void	dilate(IloNum k) const
public void	empty(IloNum xMin, IloNum xMax) const
public IloBool	everContains(const IloNumToAnySetStepFunction f) const
public IloBool	everContains(IloNum xMin, IloNum xMax, const IloAnySet elts) const
public IloBool	everContains(IloNum xMin, IloNum xMax, IloAny elt) const
public IloBool	everIntersects(const IloNumToAnySetStepFunction f) const
public IloBool	everIntersects(IloNum xMin, IloNum xMax, const IloAnySet elts) const
public void	fill(IloNum xMin, IloNum xMax) const
public IloAnySet	getComplementSet(IloNum x) const
public IloNum	getDefinitionIntervalMax() const
public IloNum	getDefinitionIntervalMin() const
public IloAnySet	getSet(IloNum x) const
public IloBool	intersects(IloNum x, const IloAnySet elts) const
public IloBool	isEmpty(IloNum x) const
public IloBool	isFull(IloNum x) const
public void	remove(const IloNumToAnySetStepFunction f) const
public void	remove(IloNum xMin, IloNum xMax, IloAny elt) const
public void	remove(IloNum xMin, IloNum xMax, const IloAnySet elts, IloBool complt=IloFalse) const
public void	set(IloNum xMin, IloNum xMax, IloAny elt) const
public void	set(IloNum xMin, IloNum xMax, const IloAnySet elts, IloBool complt=IloFalse) const
public void	setIntersection(const IloNumToAnySetStepFunction f) const
public void	setIntersection(IloNum xMin, IloNum xMax, IloAny elt) const
public void	setIntersection(IloNum xMin, IloNum xMax, const IloAnySet elts, IloBool complt=IloFalse) const

public void	setPeriodic(const IloNumToAnySetStepFunction f, IloNum x0, IloNum n, const IloAnySet dval) const
public void	shift(IloNum dx, const IloAnySet dval) const
public IloBool	usesComplementaryRepresentation(IloNum x) const

Constructors

public **IloNumToAnySetStepFunction**(const IloEnv env, IloNum xmin=-IloInfinity, IloNum xmax=IloInfinity, const char * name=0)

This constructor creates a step function defined everywhere on the interval [xMin, xMax) with empty set as the value.

public **IloNumToAnySetStepFunction**(const IloEnv env, IloNum xmin, IloNum xmax, const IloAnySet dval, const char * name=0)

This constructor creates a step function defined everywhere on the interval [xMin, xMax) with the same set dval.

Methods

public void **add**(const IloNumToAnySetStepFunction f) const

This member function adds the value of f at point x to the value of the invoking step function at point x, for all points x in the definition interval of the invoking step function. An instance of IloException is thrown if the definition interval of f is not equal to the definition interval of the invoking step function.

public void **add**(IloNum xMin, IloNum xMax, IloAny elt) const

This member function adds elt to the value of the invoking step function on the interval [xMin, xMax).

public void **add**(IloNum xMin, IloNum xMax, const IloAnySet elts, IloBool complt=IloFalse) const

This member function adds the elements of elts to the value of the invoking step function on the interval [xMin, xMax).

public IloBool **alwaysContains**(const IloNumToAnySetStepFunction f) const

This member function returns IloTrue if for all points x on the definition interval of the invoking step function, the value of f at point x is a subset of the value of the invoking step function at point x. An instance of IloException is thrown if the definition interval of f is not equal to the definition interval of the invoking step function.

public IloBool **alwaysContains**(IloNum xMin, IloNum xMax, const IloAnySet elts) const

This member function returns `IloTrue` if `elts` is a subset of the value of the invoking step function at all points on the interval `[xMin, xMax)`.

```
public IloBool alwaysContains(IloNum xMin, IloNum xMax, IloAny elt) const
```

This member function returns `IloTrue` if at all points on the interval `[xMin, xMax)` the value of the invoking step function contains `elt`.

```
public IloBool alwaysIntersects(const IloNumToAnySetStepFunction f) const
```

This member function returns `IloTrue` if for all points `x` in the definition interval of the invoking step function, the intersection of `f` and the invoking step function is not empty. An instance of `IloException` is thrown if the definition interval of `f` is not equal to the definition interval of the invoking step function.

```
public IloBool alwaysIntersects(IloNum xMin, IloNum xMax, const IloAnySet elts)
const
```

This member function returns `IloTrue` if for all `x` on the interval `[xMin, xMax)` the intersection of `elts` and the value of the invoking step function at point `x` is not empty.

```
public IloBool contains(IloNum x, const IloAnySet elts) const
```

This member function returns `IloTrue` if `elts` is a subset of the value of the invoking step function at point `x`.

```
public IloBool contains(IloNum x, IloAny elt) const
```

This member function returns `IloTrue` if the invoking step function contains element `elt` at point `x`.

```
public IloNumToAnySetStepFunction copy() const
```

This member function creates and returns a new function that is a copy of the invoking function.

```
public void dilate(IloNum k) const
```

This member function multiplies by `k` the scale of `x` for the invoking step function. `k` must be a nonnegative numeric value. More precisely, if the invoking step function was defined over an interval `[xMin, xMax)`, it will be redefined over the interval `[k*xMin, k*xMax)` and the value at `x` will be the former value at `x/k`.

```
public void empty(IloNum xMin, IloNum xMax) const
```

This member function sets the value of the invoking step function on the interval `[xMin, xMax)` to be the empty set.

```
public IloBool everContains(const IloNumToAnySetStepFunction f) const
```

This member function returns `IloTrue` if at any point `x` in the definition interval of the invoking step function, `f` at point `x` is a subset of the invoking step function at point `x`. An instance of `IloException` is thrown if the definition interval of `f` is not equal to the definition interval of the invoking step function.

`public IloBool` **`everContains`**`(IloNum xMin, IloNum xMax, const IloAnySet elts) const`

This member function returns `IloTrue` if at any point on the interval `[xMin, xMax)elts` is a subset of the value of the invoking step function.

`public IloBool` **`everContains`**`(IloNum xMin, IloNum xMax, IloAny elt) const`

This member function returns `IloTrue` if at any point on the interval `[xMin, xMax)` the value of the invoking step function contains `elt`.

`public IloBool` **`everIntersects`**`(const IloNumToAnySetStepFunction f) const`

This member function returns `IloTrue` if at some point `x` in the definition interval of the invoking step function, the intersection of `f` and the invoking step function is not empty. An instance of `IloException` is thrown if the definition interval of `f` is not equal to the definition interval of the invoking step function.

`public IloBool` **`everIntersects`**`(IloNum xMin, IloNum xMax, const IloAnySet elts) const`

This member function returns `IloTrue` if at any point `x` on the interval `[xMin, xMax)` the intersection of `elts` and the value of the invoking step function at point `x` is not empty.

`public void` **`fill`**`(IloNum xMin, IloNum xMax) const`

This member function sets the value of the invoking step function on the interval `[xMin, xMax)` to be the full set.

`public IloAnySet` **`getComplementSet`**`(IloNum x) const`

This member function returns the complement of the value of the invoking step function at point `x`. An instance of `IloException` is thrown if the invoking step function at point `x` does not use the complementary representation. See Complementary Representation of Values for more information.

`public IloNum` **`getDefinitionIntervalMax`**`() const`

This member function returns the right-most point of the definition interval of the invoking step function.

`public IloNum` **`getDefinitionIntervalMin`**`() const`

This member function returns the left-most point of the definition interval of the invoking step function.

`public IloAnySet` **`getSet`**`(IloNum x) const`

This member function returns the value of the invoking step function at point `x`. An instance of `IloException` is thrown if the invoking step function at point `x` uses the complementary representation. See Complementary Representation of Values for more information.

```
public IloBool intersects(IloNum x, const IloAnySet elts) const
```

This member function returns `IloTrue` if the intersection of `elts` and the value of the invoking step function at point `x` is not empty.

```
public IloBool isEmpty(IloNum x) const
```

This member function returns `IloTrue` if the function is empty at point `x`. In other words, a return of `IloTrue` means that the member function `IloNumToAnySetStepFunction::empty` has been applied to point `x` and no elements have been subsequently added to the value of the invoking step function at point `x`.

```
public IloBool isFull(IloNum x) const
```

This member function returns `IloTrue` if the function is full at point `x`. In other words, a return of `IloTrue` means that the member function `IloNumToAnySetStepFunction::fill` has been applied to point `x` and no elements have been subsequently removed from the value of the invoking step function at point `x`.

```
public void remove(const IloNumToAnySetStepFunction f) const
```

This member function removes the value of `f` from the value of the invoking step function at all points on the definition interval of the invoking step function. An instance of `IloException` is thrown if the definition interval of `f` is not equal to the definition interval of the invoking step function.

```
public void remove(IloNum xMin, IloNum xMax, IloAny elt) const
```

This member function removes `elt` from the value of the invoking step function on the interval `[xMin, xMax)`.

```
public void remove(IloNum xMin, IloNum xMax, const IloAnySet elts, IloBool
complt=IloFalse) const
```

This member function removes all the elements in `elts` from the value of the invoking step function on the interval `[xMin, xMax)`.

```
public void set(IloNum xMin, IloNum xMax, IloAny elt) const
```

This member function sets the value of the invoking step function to be `elt` on the interval `[xMin, xMax)`.

```
public void set(IloNum xMin, IloNum xMax, const IloAnySet elts, IloBool
complt=IloFalse) const
```

This member function sets the value of the invoking step function to be `elts` on the interval `[xMin, xMax)`.

```
public void setIntersection(const IloNumToAnySetStepFunction f) const
```

This member function assigns the value of the invoking step function at all points `x` on the definition interval of the invoking step function to be the intersection of the value of `f` at point `x` and the value of the invoking step function at point `x`. An instance of `IloException` is thrown if the definition interval of `f` is not equal to the definition interval of the invoking step function.

```
public void setIntersection(IloNum xMin, IloNum xMax, IloAny elt) const
```

This member function assigns the value of the invoking step function at all points `x` on the interval `[xMin, xMax)` to be the intersection of the set containing `elt` and the value of the invoking set function at point `x`.

```
public void setIntersection(IloNum xMin, IloNum xMax, const IloAnySet elts, IloBool complt=IloFalse) const
```

This member function assigns the value of the invoking step function at all points `x` on the interval `[xMin, xMax)` to be the intersection of `elts` and the value of the invoking set function at point `x`.

```
public void setPeriodic(const IloNumToAnySetStepFunction f, IloNum x0, IloNum n, const IloAnySet dval) const
```

This member function initializes the invoking step function as a function that repeats the step function `f`, `n` times after `x0`. More precisely, if `f` is defined on `[xfpMin,xfpMax)` and if the invoking step function is defined on `[xMin,xMax)`, the value of the invoking step function will be:

- `dval` on `[xMin, x0)`,
- `f((x-x0) % (xfpMax-xfpMin))` for `x` in `[x0, Min(x0+n*(xfpMax-xfpMin), xMax))`, and
- `dval` on `[Min(x0+n*(xfpMax-xfpMin), xMax), xMax)`

```
public void shift(IloNum dx, const IloAnySet dval) const
```

This member function shifts the invoking step function from `dx` to the right if `dx > 0`, or from `-dx` to the left if `dx < 0`. It has no effect if `dx = 0`. More precisely, if the invoking step function is defined on `[xMin,xMax)` and `dx > 0`, the new value of the invoking step function is:

- `dval` on the interval `[xMin, xMin+dx)`,
- for all `x` in [`xMin+dx, xMax)`, the former value at `x-dx`.

If `dx < 0`, the new value of the invoking step function is:

- for all `x` in `[xMin, xMax+dx)`, the former value at `x-dx`,
- `dval` on the interval `[xMax+dx, xMax)`.

```
public IloBool usesComplementaryRepresentation(IloNum x) const
```

This member function returns `IloTrue` if the value of the invoking function at point `x` is represented by a complementary set, rather than by directly representing the value as a set itself. See Complementary Representation of Values for more information.

Class IloNumToAnySetStepFunctionCursor

Definition file: ilconcert/ilosetfunc.h

Allows you to inspect the contents of an `IloNumToAnySetStepFunction`.
An instance of the class `IloNumToAnySetStepFunctionCursor` allows you to inspect the contents of an `IloNumToAnySetStepFunction`. A step of a step function is defined as an interval [x1,x2) over which the value of the function is the same. Cursors are intended to iterate forward or backward over the steps of a step function.

> **Note**
>
> The structure of the step function cannot be changed while a cursor is being used to inspect it. Therefore, functions that change the structure of the step function, such as `IloNumToAnySetStepFunction::set`, should not be called while the cursor is being used.

See Also: IloNumToAnySetStepFunction

Constructor and Destructor Summary	
`public`	`IloNumToAnySetStepFunctionCursor(const IloNumToAnySetStepFunction)`
`public`	`IloNumToAnySetStepFunctionCursor(const IloNumToAnySetStepFunction, IloNum x)`
`public`	`IloNumToAnySetStepFunctionCursor(const IloNumToAnySetStepFunctionCursor &)`

Method Summary	
`public IloAnySet`	`getComplementSet() const`
`public IloNum`	`getSegmentMax() const`
`public IloNum`	`getSegmentMin() const`
`public IloAnySet`	`getSet() const`
`public IloBool`	`isEmpty() const`
`public IloBool`	`isFull() const`
`public IloBool`	`ok() const`
`public void`	`operator++()`
`public void`	`operator--()`
`public void`	`operator=(const IloNumToAnySetStepFunctionCursor &)`
`public void`	`seek(IloNum)`
`public IloBool`	`usesComplementaryRepresentation() const`

Constructors and Destructors

`public` **`IloNumToAnySetStepFunctionCursor`**`(const IloNumToAnySetStepFunction)`

This constructor creates a cursor to inspect the step function argument. This cursor lets you iterate forward or backward over the steps of the function. The cursor initially specifies the first step of the function.

`public` **`IloNumToAnySetStepFunctionCursor`**`(const IloNumToAnySetStepFunction, IloNum x)`

This constructor creates a cursor to inspect the step function argument. This cursor lets you iterate forward or backward over the steps of the function. The cursor initially specifies the step of the function that contains `x`.

Note that if *n* is the number of steps of the function given as argument, the worst-case complexity of this constructor is *O(log(n))*.

`public` **`IloNumToAnySetStepFunctionCursor`**`(const IloNumToAnySetStepFunctionCursor &)`

This constructor creates a new cursor that is a copy of the argument. The new cursor initially specifies the same step and the same function as the argument `cursor`.

Methods

`public IloAnySet` **`getComplementSet`**`() const`

This member function returns the set representing the complement of the value of the step currently specified by the cursor. An instance of `IloException` is thrown if the value of the step does not use a complementary representation.

`public IloNum` **`getSegmentMax`**`() const`

This member function returns the right-most point of the step currently specified by the cursor.

`public IloNum` **`getSegmentMin`**`() const`

This member function returns the left-most point of the step currently specified by the cursor.

`public IloAnySet` **`getSet`**`() const`

This member function returns the value of the step currently specified by the cursor. An instance of `IloException` is thrown if the value of the step uses a complementary representation.

`public IloBool` **`isEmpty`**`() const`

This member function returns `IloTrue` if the value of the current step is the empty set.

`public IloBool` **`isFull`**`() const`

This member function returns `IloTrue` if the value of the current step is the full set. (See also: `IloNumToAnySetStepFunction::isFull`).

`public IloBool` **`ok`**`() const`

This member function returns `IloFalse` if the cursor does not currently specify a step included in the definition interval of the step function. Otherwise, it returns `IloTrue`.

```
public void operator++()
```

This operator moves the cursor to the step adjacent to the current step (forward move).

```
public void operator--()
```

This operator moves the cursor to the step adjacent to the current step (backward move).

```
public void operator=(const IloNumToAnySetStepFunctionCursor &)
```

This operator assigns an address to the handle pointer of the invoking instance of `IloNumToAnySetStepFunctionCursor`. That address is the location of the implementation object of the argument `cursor`. After the execution of this operator, the invoking object and `cursor` both point to the same implementation object.

```
public void seek(IloNum)
```

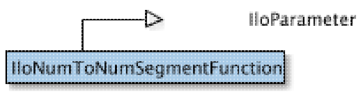
This member function sets the cursor to specify the step of the function that contains `x`. Note that if *n* is the number of steps of the step function traversed by the invoking iterator, the worst-case complexity of this member function is *O(log(n))*. An instance of `IloException` is thrown if `x` does not belong to the definition interval of the invoking function.

```
public IloBool usesComplementaryRepresentation() const
```

This member function returns `IloTrue` if the value of the current step uses the complementary representation.

Class IloNumToNumSegmentFunction

Definition file: ilconcert/ilosegfunc.h



Piecewise linear function over a segment.
An instance of `IloNumToNumSegmentFunction` represents a piecewise linear function that is defined everywhere on an interval *[xMin, xMax)*. Each interval *[x1, x2)* on which the function is linear is called a *segment*.

Note that if *n* is the number of segments of the function, the random access to a given segment (see the member functions `IloNumToNumSegmentFunction::addValue`, `IloNumToNumSegmentFunction::getArea`, `IloNumToNumSegmentFunction::getValue`, `IloNumToNumSegmentFunction::setValue`) has a worst-case complexity in *O(log(n))*.

Furthermore, when two consecutive segments of the function are co-linear, these segments are merged so that the function is always represented with the minimal number of segments.

See Also: IloNumToNumSegmentFunctionCursor

Constructor Summary	
public	IloNumToNumSegmentFunction(const IloEnv env, IloNum xmin=-IloInfinity, IloNum xmax=IloInfinity, IloNum dval=0.0, const char * name=0)
public	IloNumToNumSegmentFunction(const IloEnv env, const IloNumArray x, const IloNumArray v, IloNum xmin=-IloInfinity, IloNum xmax=IloInfinity, const char * name=0)
public	IloNumToNumSegmentFunction(const IloNumToNumStepFunction & numFunction)

Method Summary	
public void	addValue(IloNum x1, IloNum x2, IloNum v) const
public IloNumToNumSegmentFunction	copy() const
public void	dilate(IloNum k) const
public IloNum	getArea(IloNum x1, IloNum x2) const
public IloNum	getDefinitionIntervalMax() const
public IloNum	getDefinitionIntervalMin() const
public IloNum	getMax(IloNum x1, IloNum x2) const
public IloNum	getMin(IloNum x1, IloNum x2) const
public IloNum	getValue(IloNum x) const
public void	operator*=(IloNum k) const
public void	operator+=(const IloNumToNumSegmentFunction fct) const
public void	operator-=(const IloNumToNumSegmentFunction fct) const
public void	setMax(const IloNumToNumSegmentFunction fct) const
public void	setMax(IloNum x1, IloNum v1, IloNum x2, IloNum v2) const
public void	setMax(IloNum x1, IloNum x2, IloNum v) const

public void	setMin(const IloNumToNumSegmentFunction fct) const
public void	setMin(IloNum x1, IloNum v1, IloNum x2, IloNum v2) const
public void	setMin(IloNum x1, IloNum x2, IloNum v) const
public void	setPeriodic(const IloNumToNumSegmentFunction f, IloNum x0, IloNum n=IloInfinity, IloNum dval=0) const
public void	setPeriodicValue(IloNum x1, IloNum x2, const IloNumToNumSegmentFunction f, IloNum offset=0) const
public void	setPoints(const IloNumArray x, const IloNumArray v) const
public void	setSlope(IloNum x1, IloNum x2, IloNum v, IloNum slope) const
public void	setValue(IloNum x1, IloNum x2, IloNum v) const
public void	shift(IloNum dx, IloNum dval=0) const

Constructors

public **IloNumToNumSegmentFunction**(const IloEnv env, IloNum xmin=-IloInfinity, IloNum xmax=IloInfinity, IloNum dval=0.0, const char * name=0)

This constructor creates a piecewise linear function that is constant. It is defined everywhere on the interval `[xmin,xmax)` with the same value `dval`.

public **IloNumToNumSegmentFunction**(const IloEnv env, const IloNumArray x, const IloNumArray v, IloNum xmin=-IloInfinity, IloNum xmax=IloInfinity, const char * name=0)

This constructor creates a piecewise linear function defined everywhere on the interval `[xmin, xmax)` whose segments are defined by the two argument arrays `x` and `v`. More precisely, the size `n` of array `x` must be equal to the size of array `v` and, if the created function is defined on the interval `[xmin,xmax)`, its values will be:

- `v[0]` on interval `[xmin, x[0])`,
- `v[i] + (t-x[i])*(v[i+1]-v[i])/(x[i+1]-x[i])` for `t` in `[x[i], x[i+1])` for all `i` in `[0, n-2]` such that `x[i-1] <> x[i]`, and
- `v[n-1]` on interval `[x[n-1],xmax)`.

public **IloNumToNumSegmentFunction**(const IloNumToNumStepFunction & numFunction)

This copy constructor creates a new piecewise linear function. The new piecewise linear function is a copy of the step function `numFunction`. They point to different implementation objects.

Methods

public void **addValue**(IloNum x1, IloNum x2, IloNum v) const

This member function adds `v` to the value of the invoking piecewise linear function everywhere on the interval `[x1,x2)`.

`public IloNumToNumSegmentFunction **copy**() const`

This member function creates and returns a new function that is a copy of the invoking function.

`public void **dilate**(IloNum k) const`

This member function multiplies by `k` the scale of `x` for the invoking piecewise linear function. `k` must be a nonnegative numeric value. More precisely, if the invoking function was defined over an interval `[xMin,xMax)`, it will be redefined over the interval `[k*xMin,k*xMax)` and the value at `x` will be the former value at `x/k`.

`public IloNum **getArea**(IloNum x1, IloNum x2) const`

This member function returns the area of the invoking piecewise linear function on the interval `[x1,x2)`. An instance of `IloException` is thrown if the interval `[x1,x2)` is not included in the definition interval of the invoking function.

`public IloNum **getDefinitionIntervalMax**() const`

This member function returns the right-most point of the definition interval of the invoking piecewise linear function.

`public IloNum **getDefinitionIntervalMin**() const`

This member function returns the left-most point of the definition interval of the invoking piecewise linear function.

`public IloNum **getMax**(IloNum x1, IloNum x2) const`

This member function returns the maximal value of the invoking piecewise linear function on the interval `[x1,x2)`. An instance of `IloException` is thrown if the interval `[x1,x2)` is not included in the definition interval of the invoking function.

`public IloNum **getMin**(IloNum x1, IloNum x2) const`

This member function returns the minimal value of the invoking piecewise linear function on the interval `[x1,x2)`. An instance of `IloException` is thrown if the interval `[x1,x2)` is not included in the definition interval of the invoking function.

`public IloNum **getValue**(IloNum x) const`

This member function returns the value of the function at point `x`.

`public void **operator*=**(IloNum k) const`

This operator multiplies by a factor `k` the value of the invoking piecewise linear function everywhere on the definition interval.

`public void` **`operator+=`**`(const IloNumToNumSegmentFunction fct) const`

This operator adds the argument function `fct` to the invoking piecewise linear function.

`public void` **`operator-=`**`(const IloNumToNumSegmentFunction fct) const`

This operator subtracts the argument function `fct` from the invoking piecewise linear function.

`public void` **`setMax`**`(const IloNumToNumSegmentFunction fct) const`

This member function sets the value of the invoking piecewise linear function to be the maximum between the current value and the value of `fct` everywhere on the definition interval of the invoking function. The interval of definition of `fct` must be the same as that of the invoking piecewise linear function.

`public void` **`setMax`**`(IloNum x1, IloNum v1, IloNum x2, IloNum v2) const`

This member function sets the value of the invoking piecewise linear function to be the maximum between the current value and the value of the linear function:

`x --> v1 + (x-x1)*(v2-v1)/(x2-x1)` everywhere on the interval `[x1, x2)`.

`public void` **`setMax`**`(IloNum x1, IloNum x2, IloNum v) const`

This member function sets the value of the invoking piecewise linear function to be the maximum between the current value and `v` everywhere on the interval `[x1,x2)`.

`public void` **`setMin`**`(const IloNumToNumSegmentFunction fct) const`

This member function sets the value of the invoking piecewise linear function to be the minimum between the current value and the value of `fct` everywhere on the definition interval of the invoking function. The definition interval of `fct` must be the same as the one of the invoking piecewise linear function.

`public void` **`setMin`**`(IloNum x1, IloNum v1, IloNum x2, IloNum v2) const`

This member function sets the value of the invoking piecewise linear function to be the minimum between the current value and the value of the linear function:

`x --> v1 + (x-x1)*(v2-v1)/(x2-x1)` everywhere on the interval `[x1,x2)`.

`public void` **`setMin`**`(IloNum x1, IloNum x2, IloNum v) const`

This member function sets the value of the invoking piecewise linear function to be the minimum between the current value and `v` everywhere on the interval `[x1,x2)`.

`public void setPeriodic(const IloNumToNumSegmentFunction f, IloNum x0, IloNum n=IloInfinity, IloNum dval=0) const`

This member function initializes the invoking function as a piecewise linear function that repeats the piecewise linear function `f`, `n` times after `x0`. More precisely, if `f` is defined on `[xfpMin,xfpMax)` and if the invoking function is defined on `[xMin,xMax)`, the value of the invoking function will be:

- `dval` on `[xMin, x0)`,
- `f((x-x0) % (xfpMax-xfpMin))` for `x` in `[x0, Min(x0+n*(xfpMax-xfpMin), xMax))`, and
- `dval` on `[Min(x0+n*(xfpMax-xfpMin), xMax), xMax)`

`public void setPeriodicValue(IloNum x1, IloNum x2, const IloNumToNumSegmentFunction f, IloNum offset=0) const`

This member function changes the value of the invoking function on the interval `[x1,x2)`. On this interval, the invoking function is set to equal a repetition of the pattern function `f` with an initial offset of `offset`. The invoking function is not modified outside the interval `[x1,x2)`. More precisely, if `[min,max)` denotes the definition interval of `f`, for all `t` in `[x1,x2)`, the invoking function at `t` is set to equal `f(min + (offset+t-x1)%(max-min)))` where `%` denotes the modulo operator. By default, the offset is equal to 0.

`public void setPoints(const IloNumArray x, const IloNumArray v) const`

This member function initializes the invoking function as a piecewise linear function whose segments are defined by the two argument arrays `x` and `v`.

More precisely, the size `n` of array `x` must be equal to the size of array `v`, and if the created function is defined on the interval `[xmin,xmax)`, its values will be:

- `v[0]` on interval `[xmin, x[0])`,
- `v[i] + (t-x[i])*(v[i+1]-v[i])/(x[i+1]-x[i])` for `t` in `[x[i], x[i+1])` for all `i` in `[0, n-2]` such that `x[i-1] ≠ x[i]`, and
- `v[n-1]` on interval `[x[n-1],xmax)`.

`public void setSlope(IloNum x1, IloNum x2, IloNum v, IloNum slope) const`

This member function sets the value of the invoking piecewise linear function equal to `f`, associating for each `x` in `[x1,x2) -> f(x) = v + slope * (x-x1)`.

`public void setValue(IloNum x1, IloNum x2, IloNum v) const`

This member function sets the value of the invoking piecewise linear function to be constant and equal to `v` on the interval `[x1,x2)`.

`public void shift(IloNum dx, IloNum dval=0) const`

This member function shifts the invoking function from `dx` to the right if `dx > 0` or `-dx` to the left if `dx < 0`. It has no effect if `dx = 0`. More precisely, if the invoking function is defined on `[xMin,xMax)` and `dx > 0`, the new value of the invoking function is:

- `dval` on the interval `[xMin,xMin+dx)`,

- for all `x` in `[xMin+dx,xMax)`, the former value at `x-dx`.

If `dx < 0`, the new value of the invoking function is:

- for all `x` in `[xMin,xMax+dx)`, the former value at `x-dx`,
- `dval` on the interval `[xMax+dx,xMax)`.

Class IloNumToNumSegmentFunctionCursor

Definition file: ilconcert/ilosegfunc.h

Cursor over segments of a piecewise linear function.
An instance of the class `IloNumToNumSegmentFunctionCursor` allows you to inspect the contents of an `IloNumToNumSegmentFunction`. A segment of a piecewise linear function is defined as an interval [`x1`, `x2`) over which the function is linear. Cursors are intended to iterate forward or backward over the segments of a piecewise linear function.

> **Note**
>
> The structure of the piecewise linear function cannot be changed while a cursor is being used to inspect it. Therefore, functions that change the structure of the piecewise linear function, such as `IloNumToNumStepFunction::setValue`, should not be called while the cursor is being used.

See Also: IloNumToNumSegmentFunction

Constructor and Destructor Summary	
`public`	`IloNumToNumSegmentFunctionCursor(const IloNumToNumSegmentFunction, IloNum x)`
`public`	`IloNumToNumSegmentFunctionCursor(const IloNumToNumSegmentFunctionCursor &)`

Method Summary	
`public IloNum`	`getSegmentMax() const`
`public IloNum`	`getSegmentMin() const`
`public IloNum`	`getValue(IloNum t) const`
`public IloNum`	`getValueLeft() const`
`public IloNum`	`getValueRight() const`
`public IloBool`	`ok() const`
`public void`	`operator++()`
`public void`	`operator--()`
`public void`	`seek(IloNum)`

Constructors and Destructors

`public` **`IloNumToNumSegmentFunctionCursor`**`(const IloNumToNumSegmentFunction, IloNum x)`

This constructor creates a cursor to inspect the piecewise linear function argument. This cursor lets you iterate forward or backward over the segments of the function. The cursor initially specifies the segment of the function that contains `x`.

Note that if *n* is the number of steps of the function given as argument, the worst-case complexity of this constructor is *O(log(n))*.

`public` **`IloNumToNumSegmentFunctionCursor`**`(const IloNumToNumSegmentFunctionCursor &)`

This constructor creates a new cursor that is a copy of the argument `cursor`. The new cursor initially specifies the same segment and the same function as the argument `cursor`.

Methods

`public IloNum` **`getSegmentMax`**`() const`

This member function returns the right-most point of the segment currently specified by the cursor.

`public IloNum` **`getSegmentMin`**`() const`

This member function returns the left-most point of the segment currently specified by the cursor.

`public IloNum` **`getValue`**`(IloNum t) const`

This member function returns the value of the piecewise linear function at time `t`. `t` must be between the left-most and the right-most point of the segment currently specified by the cursor.

`public IloNum` **`getValueLeft`**`() const`

This member function returns the value of the function at the left-most point of the segment currently specified by the cursor.

`public IloNum` **`getValueRight`**`() const`

This member function returns the value of the function at the right-most point of the segment currently specified by the cursor.

`public IloBool` **`ok`**`() const`

This member function returns `IloFalse` if the cursor does not currently specify a segment included in the definition interval of the piecewise linear function. Otherwise, it returns `IloTrue`.

`public void` **`operator++`**`()`

This operator moves the cursor to the segment adjacent to the current step (forward move).

`public void` **`operator--`**`()`

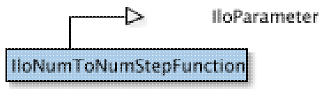
This operator moves the cursor to the segment adjacent to the current step (backward move).

`public void` **`seek`**`(IloNum)`

This member function sets the cursor to specify the segment of the function that contains the argument. An `IloException` is thrown if the argument does not belong to the definition interval of the piecewise linear function associated with the invoking cursor.

Class IloNumToNumStepFunction

Definition file: ilconcert/ilonumfunc.h



Represents a step function that is defined everywhere on an interval.
An instance of `IloNumToNumStepFunction` represents a step function that is defined everywhere on an interval [`xMin`, `xMax`). Each interval [`x1`, `x2`) on which the function has the same value is called a step.

Note that if *n* is the number of steps of the function, the random access to a given step (see the member functions `IloNumToNumStepFunction::addValue`, `IloNumToNumStepFunction::getArea`, `IloNumToNumStepFunction::getValue`, `IloNumToNumStepFunction::setValue`) has a worst-case complexity in *O(log(n))*.

Furthermore, when two consecutive steps of the function have the same value, these steps are merged so that the function is always represented with the minimal number of steps.

See Also: IloNumToNumStepFunctionCursor

Constructor Summary	
public	IloNumToNumStepFunction(const IloEnv env, IloNum xmin=-IloInfinity, IloNum xmax=IloInfinity, IloNum dval=0.0, const char * name=0)
public	IloNumToNumStepFunction(const IloEnv env, const IloNumArray x, const IloNumArray v, IloNum xmin=-IloInfinity, IloNum xmax=IloInfinity, const char * name=0)

Method Summary	
public void	addValue(IloNum x1, IloNum x2, IloNum v) const
public IloNumToNumStepFunction	copy() const
public void	dilate(IloNum k) const
public IloNum	getArea(IloNum x1, IloNum x2) const
public IloNum	getDefinitionIntervalMax() const
public IloNum	getDefinitionIntervalMin() const
public IloNum	getMax(IloNum x1, IloNum x2) const
public IloNum	getMin(IloNum x1, IloNum x2) const
public IloNum	getValue(IloNum x) const
public void	operator*=(IloNum k) const
public void	operator+=(const IloNumToNumStepFunction fct) const
public void	operator-=(const IloNumToNumStepFunction fct) const
public void	setMax(const IloNumToNumStepFunction fct) const
public void	setMax(IloNum x1, IloNum x2, IloNum v) const
public void	setMin(const IloNumToNumStepFunction fct) const
public void	setMin(IloNum x1, IloNum x2, IloNum v) const
public void	setPeriodic(const IloNumToNumStepFunction f, IloNum x0, IloNum n=IloInfinity, IloNum dval=0) const
public void	

	setPeriodicValue(IloNum x1, IloNum x2, const IloNumToNumStepFunction f, IloNum offset=0) const
public void	setSteps(const IloNumArray x, const IloNumArray v) const
public void	setValue(IloNum x1, IloNum x2, IloNum v) const
public void	shift(IloNum dx, IloNum dval=0) const

Constructors

public **IloNumToNumStepFunction**(const IloEnv env, IloNum xmin=-IloInfinity, IloNum xmax=IloInfinity, IloNum dval=0.0, const char * name=0)

This constructor creates a step function defined everywhere on the interval `[xmin, xmax)` with the same value `dval`.

public **IloNumToNumStepFunction**(const IloEnv env, const IloNumArray x, const IloNumArray v, IloNum xmin=-IloInfinity, IloNum xmax=IloInfinity, const char * name=0)

This constructor creates a step function defined everywhere on the interval `[xmin, xmax)` whose steps are defined by the two argument arrays `x` and `v`. More precisely, if `n` is the size of array `x`, size of array `v` must be `n+1` and, if the created function is defined on the interval `[xmin,xmax)`, its values will be:

- `v[0]` on interval `[xmin,x[0])`,
- `v[i]` on interval `[x[i-1],x[i])` for all `i` in `[0,n-1]`, and
- `v[n]` on interval `[x[n-1],xmax)`.

The values in the array are copied, and no modification to the arrays will be taken into account once the constructor has been called.

Methods

public void **addValue**(IloNum x1, IloNum x2, IloNum v) const

This member function adds `v` to the value of the invoking step function everywhere on the interval `[x1, x2)`.

public IloNumToNumStepFunction **copy**() const

This member function creates and returns a new function that is a copy of the invoking function.

public void **dilate**(IloNum k) const

This member function multiplies by `k` the scale of `x` for the invoking step function. `k` must be a nonnegative numeric value. More precisely, if the invoking function was defined over an interval `[xMin, xMax)`, it will be redefined over the interval `[k*xMin, k*xMax)` and the value at `x` will be the former value at `x/k`.

public IloNum **getArea**(IloNum x1, IloNum x2) const

This member function returns the sum of the invoking step function on the interval `[x1, x2)`. An instance of `IloException` is thrown if the interval `[x1, x2)` is not included in the definition interval of the invoking function.

`public IloNum` **`getDefinitionIntervalMax`**`() const`

This member function returns the right-most point of the definition interval of the invoking step function.

`public IloNum` **`getDefinitionIntervalMin`**`() const`

This member function returns the left-most point of the definition interval of the invoking step function.

`public IloNum` **`getMax`**`(IloNum x1, IloNum x2) const`

This member function returns the maximal value of the invoking step function on the interval `[x1, x2)`. An instance of `IloException` is thrown if the interval `[x1, x2)` is not included in the definition interval of the invoking function.

`public IloNum` **`getMin`**`(IloNum x1, IloNum x2) const`

This member function returns the minimal value of the invoking step function on the interval `[x1, x2)`. An instance of `IloException` is thrown if the interval `[x1, x2)` is not included in the definition interval of the invoking function.

`public IloNum` **`getValue`**`(IloNum x) const`

This member function returns the value of the invoking step function at `x`. An instance of `IloException` is thrown if `x` does not belong to the definition interval of the invoking function.

`public void` **`operator*=`**`(IloNum k) const`

This operator multiplies by a factor `k` the value of the invoking step function everywhere on the definition interval.

`public void` **`operator+=`**`(const IloNumToNumStepFunction fct) const`

This operator adds the argument function `fct` to the invoking step function.

`public void` **`operator-=`**`(const IloNumToNumStepFunction fct) const`

This operator subtracts the argument function `fct` from the invoking step function.

`public void` **`setMax`**`(const IloNumToNumStepFunction fct) const`

This member function sets the value of the invoking step function to be the maximum between the current value and the value of `fct` everywhere on the definition interval of the invoking function. The interval of definition of `fct` must be the same as that of the invoking step function.

```
public void setMax(IloNum x1, IloNum x2, IloNum v) const
```

This member function sets the value of the invoking step function to be the maximum between the current value and `v` everywhere on the interval `[x1, x2)`.

```
public void setMin(const IloNumToNumStepFunction fct) const
```

This member function sets the value of the invoking step function to be the minimum between the current value and the value of `fct` everywhere on the definition interval of the invoking function. The definition interval of `fct` must be the same as the one of the invoking step function.

```
public void setMin(IloNum x1, IloNum x2, IloNum v) const
```

This member function sets the value of the invoking step function to be the minimum between the current value and `v` everywhere on the interval `[x1, x2)`.

```
public void setPeriodic(const IloNumToNumStepFunction f, IloNum x0, IloNum
n=IloInfinity, IloNum dval=0) const
```

This member function initializes the invoking function as a step function that repeats the step function `f`, `n` times after `x0`. More precisely, if `f` is defined on `[xfpMin,xfpMax)` and if the invoking function is defined on `[xMin,xMax)`, the value of the invoking function will be:

- `dval` on `[xMin, x0)`,
- `fp((x-x0) % (xfpMax-xfpMin))` for `x` in `[x0, Min(x0+n*(xfpMax-xfpMin),xMax))`, and
- `dval` on `[Min(x0+n*(xfpMax-xfpMin), xMax), xMax)`

```
public void setPeriodicValue(IloNum x1, IloNum x2, const IloNumToNumStepFunction f,
IloNum offset=0) const
```

This member function changes the value of the invoking function on the interval `[x1,x2)`. On this interval, the invoking function is set to equal a repetition of the pattern function `f` with an initial offset of `offset`. The invoking function is not modified outside the interval `[x1,x2)`. More precisely, if `[min,max)` denotes the definition interval of `f`, for all `t` in `[x1,x2)`, the invoking function at `t` is set to equal `f(min + (offset+t-x1)%(max-min)))` where `%` denotes the modulo operator. By default, the offset is equal to 0.

```
public void setSteps(const IloNumArray x, const IloNumArray v) const
```

This member function initializes the invoking function as a step function whose steps are defined by the two arguments arrays `x` and `v`. More precisely, if `n` is the size of array `x`, size of array `v` must be `n+1` and, if the invoking function is defined on the interval `[xMin,xMax)`, its values will be:

- `v[0]` on interval `[xMin,x[0])`,
- `v[i]` on interval `[x[i-1],x[i])` for all `i` in `[0,n-1]`, and
- `v[n]` on interval `[x[n-1],xMax)`.

`public void setValue(IloNum x1, IloNum x2, IloNum v) const`

This member function sets the value of the invoking step function to be `v` on the interval `[x1, x2)`.

`public void shift(IloNum dx, IloNum dval=0) const`

This member function shifts the invoking function from `dx` to the right if `dx > 0` or from `-dx` to the left if `dx < 0`. It has no effect if `dx = 0`. More precisely, if the invoking function is defined on `[xMin,xMax)` and `dx > 0`, the new value of the invoking function is:

- `dval` on the interval `[xMin, xMin+dx)`,
- for all `x` in `[xMin+dx, xMax)`, the former value at `x-dx`.

If `dx < 0`, the new value of the invoking function is:

- for all `x` in `[xMin, xMax+dx)`, the former value at `x-dx`,
- `dval` on the interval `[xMax+dx,xMax)`.

Class IloNumToNumStepFunctionCursor

Definition file: ilconcert/ilonumfunc.h

Allows you to inspect the contents of an instance of IloNumToNumStepFunction.
An instance of the class `IloNumToNumStepFunctionCursor` allows you to inspect the contents of an instance of `IloNumToNumStepFunction`. A step of a step function is defined as an interval [x1,x2) over which the value of the function is the same. Cursors are intended to iterate forward or backward over the steps of a step function.

> **Note**
>
> The structure of the step function cannot be changed while a cursor is being used to inspect it. Therefore, methods that change the structure of the step function, such as `IloNumToNumStepFunction::setValue`, should not be called while the cursor is being used.

See Also: IloNumToNumStepFunction

Constructor and Destructor Summary	
`public`	`IloNumToNumStepFunctionCursor(const IloNumToNumStepFunction, IloNum x)`
`public`	`IloNumToNumStepFunctionCursor(const IloNumToNumStepFunctionCursor &)`

Method Summary	
`public IloNum`	`getSegmentMax() const`
`public IloNum`	`getSegmentMin() const`
`public IloNum`	`getValue() const`
`public IloBool`	`ok() const`
`public void`	`operator++()`
`public void`	`operator--()`
`public void`	`seek(IloNum)`

Constructors and Destructors

`public` **`IloNumToNumStepFunctionCursor`**`(const IloNumToNumStepFunction, IloNum x)`

This constructor creates a cursor to inspect the step function argument. This cursor lets you iterate forward or backward over the steps of the function. The cursor initially specifies the step of the function that contains `x`.

Note that if *n* is the number of steps of the function given as argument, the worst-case complexity of this constructor is *O(log(n))*.

`public` **`IloNumToNumStepFunctionCursor`**`(const IloNumToNumStepFunctionCursor &)`

This constructor creates a new cursor that is a copy of the argument `cursor`. The new cursor initially specifies the same step and the same function as the argument `cursor`.

Methods

`public IloNum getSegmentMax() const`

This member function returns the right-most point of the step currently specified by the cursor.

`public IloNum getSegmentMin() const`

This member function returns the left-most point of the step currently specified by the cursor.

`public IloNum getValue() const`

This member function returns the value of the step currently specified by the cursor.

`public IloBool ok() const`

This member function returns `IloFalse` if the cursor does not currently specify a step included in the definition interval of the step function. Otherwise, it returns `IloTrue`.

`public void operator++()`

This operator moves the cursor to the step adjacent to the current step (forward move).

`public void operator--()`

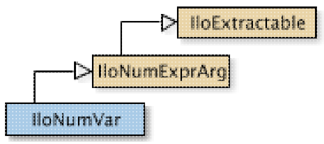
This operator moves the cursor to the step adjacent to the current step (backward move).

`public void seek(IloNum)`

This member function sets the cursor to specify the step of the function that contains the argument. An `IloException` is thrown if the argument does not belong to the definition interval of the step function associated with the invoking cursor.

Class IloNumVar

Definition file: ilconcert/iloexpression.h



An instance of this class represents a numeric variable in a model.
An instance of this class represents a numeric variable in a model. A numeric variable may be either an integer variable or a floating-point variable; that is, a numeric variable has a type, a value of the nested enumeration `IloNumVar::Type`. By default, its type is `Float`. It also has a lower and upper bound. A numeric variable cannot assume values less than its lower bound, nor greater than its upper bound.

If you are looking for a class of variables that can assume only constrained integer values, consider the class `IloIntVar`. If you are looking for a class of binary decision variables that can assume only the values 0 (zero) or 1 (one), then consider the class `IloBoolVar`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Programming Hint

For each enumerated value in the nested enumeration `IloNumVar::Type`, Concert Technology offers an equivalent predefined C++ `#define` to make programming easier. For example, in your applications, you may write either statement:

```
IloNumVar x(env, 0, 17, IloNumVar::Int); // using the enumeration

IloNumVar x(env, 0, 17, ILOINT);         // using the#define
```

> **Note**
>
> When numeric bounds are given to an integer variable (an instance of `IloIntVar` or `IloNumVar` with `Type = Int`) in the constructors or via a modifier (`setUB`, `setLB`, `setBounds`), they are inwardly rounded to an integer value. `LB` is rounded down and `UB` is rounded up.

See Also: IloBoolVar, IloIntVar, IloModel, IloNumVarArray, IloNumVar::Type

Constructor Summary	
`public`	`IloNumVar()`
`public`	`IloNumVar(IloNumVarI * impl)`
`public`	`IloNumVar(const IloEnv env, IloNum lb=0, IloNum ub=IloInfinity, IloNumVar::Type type=Float, const char * name=0)`
`public`	`IloNumVar(const IloEnv env, IloNum lowerBound, IloNum upperBound, const char * name)`
`public`	`IloNumVar(const IloAddNumVar & var, IloNum lowerBound=0.0, IloNum upperBound=IloInfinity, IloNumVar::Type type=Float, const char * name=0)`
`public`	`IloNumVar(const IloEnv env, const IloNumArray values, IloNumVar::Type type=Float, const char * name=0)`
`public`	`IloNumVar(const IloAddNumVar & var, const IloNumArray values, IloNumVar::Type type=Float, const char * name=0)`

public	IloNumVar(const IloConstraint constraint)
public	IloNumVar(const IloNumRange coll, const char * name=0)

Method Summary	
public IloNumVarI *	getImpl() const
public IloNum	getLB() const
public void	getPossibleValues(IloNumArray values) const
public IloNumVar::Type	getType() const
public IloNum	getUB() const
public void	setBounds(IloNum lb, IloNum ub) const
public void	setLB(IloNum num) const
public void	setPossibleValues(const IloNumArray values) const
public void	setUB(IloNum num) const

Inherited Methods from `IloNumExprArg`
getImpl

Inherited Methods from `IloExtractable`
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Inner Enumeration	
IloNumVar::Type	An enumeration for the class IloNumVar.

Constructors

public **IloNumVar**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IloNumVar**(IloNumVarI * impl)

This constructor creates a handle object from a pointer to an implementation object.

public **IloNumVar**(const IloEnv env, IloNum lb=0, IloNum ub=IloInfinity, IloNumVar::Type type=Float, const char * name=0)

This constructor creates a constrained numeric variable and makes it part of the environment `env`. By default, the numeric variable ranges from 0.0 (zero) to the symbolic constant `IloInfinity`, but you can specify other upper and lower bounds yourself. By default, the numeric variable assumes floating-point values. However, you can constrain it to be an integer by setting its `type = Int`. By default, its name is the empty string, but you can specify a `name` of your own choice.

public **IloNumVar**(const IloEnv env, IloNum lowerBound, IloNum upperBound, const char * name)

This constructor creates a constrained numeric variable and makes it part of the environment `env`. The bounds of the variable are set by `lowerBound` and `upperBound`. By default, its name is the empty string, but you can specify a `name` of your own choice.

`public IloNumVar(const IloAddNumVar & var, IloNum lowerBound=0.0, IloNum upperBound=IloInfinity, IloNumVar::Type type=Float, const char * name=0)`

This constructor creates a constrained numeric variable in column format. For more information on adding columns to a model, refer to the concept *Column-Wise Modeling*.

`public IloNumVar(const IloEnv env, const IloNumArray values, IloNumVar::Type type=Float, const char * name=0)`

This constructor creates a constrained discrete numeric variable and makes it part of the environment `env`. The new discrete variable will be limited to values in the set specified by the array `values`. By default, its name is the empty string, but you can specify a name of your own choice. You can use the member function `IloNumVar::setPossibleValues` with instances created by this constructor.

`public IloNumVar(const IloAddNumVar & var, const IloNumArray values, IloNumVar::Type type=Float, const char * name=0)`

This constructor creates a constrained *discrete* numeric variable from `var` by limiting its domain to the values specified in the array `values`. You may use the member function `IloNumVar::setPossibleValues` with instances created by this constructor.

`public IloNumVar(const IloConstraint constraint)`

This constructor creates a constrained numeric variable which is equal to the truth value of `constraint`. The truth value of `constraint` is either 0 for `IloFalse` or 1 for `IloTrue`. You can use this constructor to cast a constraint to a constrained numeric variable. That constrained numeric variable can then be used like any other constrained numeric variable. It is thus possible to express sums of constraints, for example. The following line expresses the idea that all three variables cannot be equal:

```
model.add((x != y) + (y != z) + (z != x) >= 2);
```

`public IloNumVar(const IloNumRange coll, const char * name=0)`

This constructor creates a constrained *discrete* numeric variable from the given collection

This constructor creates a constrained *discrete* numeric variable from the given collection

Methods

`public IloNumVarI * getImpl() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public IloNum getLB() const`

This member function returns the lower bound of the invoking numeric variable.

`public void getPossibleValues(IloNumArray values) const`

This member function accesses the possible values of the invoking numeric variable and puts them in the array `values`.

`public IloNumVar::Type **getType**() const`

This member function returns the type of the invoking numeric variable, specifying whether it is integer (`Int`) or floating-point (`Float`).

`public IloNum **getUB**() const`

This member function returns the upper bound of the invoking numeric variable.

`public void **setBounds**(IloNum lb, IloNum ub) const`

This member function sets `lb` as the lower bound and `ub` as the upper bound of the invoking numeric variable.

Note

The member function `setBounds` notifies Concert Technology algorithms about this change of bounds in this numeric variable.

`public void **setLB**(IloNum num) const`

This member function sets `num` as the lower bound of the invoking numeric variable.

Note

The member function `setLB` notifies Concert Technology algorithms about this change of bound in this numeric variable.

`public void **setPossibleValues**(const IloNumArray values) const`

This member function sets `values` as the domain of the invoking discrete numeric variable. This member function can be called only on instances of `IloNumVar` that have been created with one of the two *discrete* constructors; that is, instances of `IloNumVar` which have been defined by an explicit array of discrete values.

Note

The member function `setPossibleValues` notifies Concert Technology algorithms about this change of domain in this discrete numeric variable.

`public void **setUB**(IloNum num) const`

This member function sets `num` as the upper bound of the invoking numeric variable.

Note

The member function `setUB` notifies Concert Technology algorithms about this change of bound in this numeric variable.

Inner Enumerations

Enumeration Type

Definition file: ilconcert/iloexpression.h

An enumeration for the class IloNumVar.
This nested enumeration enables you to specify whether an instance of `IloNumVar` is of type integer (`Int`), Boolean (`Bool`), or floating-point (`Float`).

Programming Hint

For each enumerated value in `IloNumVar::Type`, there is a predefined equivalent C++ `#define` in the Concert Technology include files to make programming easier. For example, instead of writing `IloNumVar::Int` in your application, you can write `ILOINT`. Likewise, `ILOFLOAT` is defined for `IloNumVar::Float` and `ILOBOOL` for `IloNumVar::Bool`.

See Also: IloNumVar

Fields:

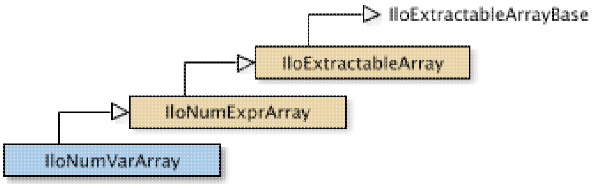
`Int = 1`

`Float = 2`

`Bool = 3`

Class IloNumVarArray

Definition file: ilconcert/iloexpression.h



The array class of IloNumVar.
For each basic type, Concert Technology defines a corresponding array class. `IloNumVarArray` is the array class of the numeric variable class for a model.

Instances of `IloNumVarArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloAllDiff, IloModel, IloNumVar, operator>>, operator<<

Constructor Summary	
public	IloNumVarArray(IloDefaultArrayI * i=0)
public	IloNumVarArray(const IloEnv env, IloInt n=0)
public	IloNumVarArray(const IloEnv env, const IloNumArray lb, const IloNumArray ub, IloNumVar::Type type=ILOFLOAT)
public	IloNumVarArray(const IloEnv env, IloNum lb, const IloNumArray ub, IloNumVar::Type type=ILOFLOAT)
public	IloNumVarArray(const IloEnv env, const IloNumArray lb, IloNum ub, IloNumVar::Type type=ILOFLOAT)
public	IloNumVarArray(const IloEnv env, IloInt n, IloNum lb, IloNum ub, IloNumVar::Type type=ILOFLOAT)
public	IloNumVarArray(const IloEnv env, const IloNumColumnArray columnarray, IloNumVar::Type type=ILOFLOAT)
public	IloNumVarArray(const IloEnv env, const IloNumColumnArray columnarray, const IloNumArray lb, const IloNumArray ub, IloNumVar::Type type=ILOFLOAT)

Method Summary	
public void	add(IloInt more, const IloNumVar x)
public void	add(const IloNumVar x)
public void	add(const IloNumVarArray array)
public void	endElements()
public IloNumExprArg	operator[](IloIntExprArg anIntegerExpr) const
public void	setBounds(const IloNumArray lb, const IloNumArray ub)
public IloIntExprArray	toIntExprArray() const
public IloIntVarArray	toIntVarArray() const
public IloNumExprArray	toNumExprArray() const

Inherited Methods from `IloNumExprArray`
add, add, add, endElements, operator[]

Inherited Methods from `IloExtractableArray`
add, add, add, endElements, setNames

Constructors

public **IloNumVarArray**(IloDefaultArrayI * i=0)

This constructor creates an empty extensible array of numeric variables. You cannot create instances of the undocumented class `IloDefaultArrayI`. As an argument in this default constructor, it allows you to pass 0 (zero) optionally or to use 0 (zero) as a default value of an argument in a constructor.

public **IloNumVarArray**(const IloEnv env, IloInt n=0)

This constructor creates an extensible array of `n` numeric variables in `env`. Initially, the `n` elements are empty handles.

public **IloNumVarArray**(const IloEnv env, const IloNumArray lb, const IloNumArray ub, IloNumVar::Type type=ILOFLOAT)

This constructor creates an extensible array of numeric variables in `env` with lower and upper bounds and type as specified. The instances of `IloNumVar` to fill this array are constructed at the same time. The length of the array `lb` must be the same as the length of the array `ub`. In other words, `lb.getSize == ub.getSize`. This constructor will construct an array with the number of elements in the array `ub`.

public **IloNumVarArray**(const IloEnv env, IloNum lb, const IloNumArray ub, IloNumVar::Type type=ILOFLOAT)

This constructor creates an extensible array of numeric variables in `env` with lower and upper bounds and type as specified. The instances of `IloNumVar` to fill this array are constructed at the same time. The length of the new array will be the same as the length of the array `ub`.

public **IloNumVarArray**(const IloEnv env, const IloNumArray lb, IloNum ub, IloNumVar::Type type=ILOFLOAT)

This constructor creates an extensible array of numeric variables in `env` with lower and upper bounds and type as specified. The instances of `IloNumVar` to fill this array are constructed at the same time.

public **IloNumVarArray**(const IloEnv env, IloInt n, IloNum lb, IloNum ub, IloNumVar::Type type=ILOFLOAT)

This constructor creates an extensible array of `n` numeric variables in `env` with lower and upper bounds and type as specified. The instances of `IloNumVar` to fill this array are constructed at the same time.

public **IloNumVarArray**(const IloEnv env, const IloNumColumnArray columnarray, IloNumVar::Type type=ILOFLOAT)

This constructor creates an extensible array of numeric variables with type as specified. The instances of `IloNumVar` to fill this array are constructed at the same time.

`public IloNumVarArray(const IloEnv env, const IloNumColumnArray columnarray, const IloNumArray lb, const IloNumArray ub, IloNumVar::Type type=ILOFLOAT)`

This constructor creates an extensible array of numeric variables with lower and upper bounds and type as specified. The instances of `IloNumVar` to fill this array are constructed at the same time.

Methods

`public void add(IloInt more, const IloNumVar x)`

This member function appends `x` to the invoking array multiple times. The argument `more` specifies how many times.

`public void add(const IloNumVar x)`

This member function appends `x` to the invoking array.

`public void add(const IloNumVarArray array)`

This member function appends the elements in `array` to the invoking array.

`public void endElements()`

This member function calls `IloExtractable::end` for each of the elements in the invoking array. This deletes all the extractables identified by the array, leaving the handles in the array intact. This member function is the recommended way to delete the elements of an array.

`public IloNumExprArg operator[](IloIntExprArg anIntegerExpr) const`

This subscripting operator returns an expression argument for use in a constraint or expression. For clarity, let's call `A` the invoking array. When `anIntegerExpr` is bound to the value `i`, the domain of the expression is the domain of `A[i]`. More generally, the domain of the expression is the union of the domains of the expressions `A[i]` where the `i` are in the domain of `anIntegerExpr`.

This operator is also known as an element expression.

`public void setBounds(const IloNumArray lb, const IloNumArray ub)`

For each element in the invoking array, this member function sets `lb` as the lower bound and `ub` as the upper bound of the corresponding numeric variable in the invoking array.

Note

The member function `setBounds` notifies Concert Technology algorithms about this change of bounds in this array of numeric variables.

```
public IloIntExprArray toIntExprArray() const
```

This member function copies the invoking array to a new `IloIntExprArray`, checking the type of the variables during the copy.

```
public IloIntVarArray toIntVarArray() const
```

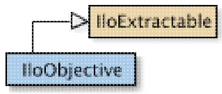
This member function copies the invoking array to a new `IloIntVarArray`, checking the type of the variables during the copy.

```
public IloNumExprArray toNumExprArray() const
```

This member function copies the invoking array to a new `IloNumExprArray`, checking the type of the variables during the copy.

Class IloObjective

Definition file: ilconcert/ilolinear.h



An instance of this class is an objective in a model.
An objective consists of its sense (specifying whether it is a minimization or maximization) and an expression. The expression may be a constant.

An objective belongs to the environment that the variables in its expression belong to. Generally, you will create an objective, add it to a model, and extract the model for an algorithm.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

What Is Extracted

All the variables (that is, instances of `IloNumVar` or one of its subclasses) in the objective (an instance of `IloObjective`) will be extracted when an algorithm such as `IloCplex`, documented in the *IBM ILOG CPLEX Reference Manual*, extracts the objective.

Multiple Objectives

You may create more than one objective in a model, for example, by creating more than one group. However, certain algorithms, such as an instance of `IloCplex`, will throw an exception (on a platform that supports C++ exceptions, when exceptions are enabled) if you attempt to extract more than one objective at a time.

Also see the functions `IloMaximize` and `IloMinimize` for “short cuts” to create objectives.

Normalizing Linear Expressions: Reducing the Terms

Normalizing is sometimes known as *reducing the terms* of a linear expression.

Linear expressions consist of terms made up of constants and variables related by arithmetic operations; for example, x + 3y is a linear expression of two terms consisting of two variables. In some linear expressions, a given variable may appear in more than one term, for example, x + 3y +2x. Concert Technology has more than one way of dealing with linear expressions in this respect, and you control which way Concert Technology treats linear expressions from your application.

In one mode (the default mode), Concert Technology analyzes expressions that your application passes it and attempts to reduce them so that a given variable appears in only one term in the expression. You set this mode with the member function `IloEnv::setNormalizer`.

Certain constructors and member functions in this class check this setting in the model and behave accordingly: they attempt to reduce expressions. This mode may require more time during preliminary computation, but it avoids the possibility of an assertion failing for certain member functions of this class in case of duplicates.

In the other mode, Concert Technology **assumes** that no variable appears in more than one term in any of the linear expressions that your application passes to Concert Technology. We call this mode assume no duplicates. You set this mode with the member function `IloEnv::setNormalizer`.

Certain constructors and member functions in this class check this setting in the model and behave accordingly: they assume that no variable appears in more than one term in an expression. This mode may save time during computation, but it entails the risk that an expression may contain one or more variables, each of which appears in one or more terms. This situation will cause certain `assert` statements in Concert Technology to fail if you do not compile with the flag `-DNDEBUG`.

See Also: IloMaximize, IloMinimize, IloModel, IloObjective::Sense

Constructor Summary	
public	`IloObjective()`
public	`IloObjective(IloObjectiveI * impl)`
public	`IloObjective(const IloEnv env, IloNum constant=0.0, IloObjective::Sense sense=Minimize, const char * name=0)`
public	`IloObjective(const IloEnv env, const IloNumExprArg expr, IloObjective::Sense sense=Minimize, const char * name=0)`

Method Summary	
`public IloNum`	`getConstant() const`
`public IloNumExprArg`	`getExpr() const`
`public IloObjectiveI *`	`getImpl() const`
`public IloObjective::Sense`	`getSense() const`
`public IloAddValueToObj`	`operator()(IloNum value)`
`public IloAddValueToObj`	`operator()()`
`public void`	`setConstant(IloNum constant)`
`public void`	`setExpr(const IloNumExprArg)`
`public void`	`setLinearCoef(const IloNumVar var, IloNum value)`
`public void`	`setLinearCoefs(const IloNumVarArray vars, const IloNumArray values)`
`public void`	`setSense(IloObjective::Sense sense)`

Inherited Methods from `IloExtractable`
`asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject`

Inner Enumeration	
IloObjective::Sense	Specifies objective as minimization or maximization.

Constructors

`public` **`IloObjective`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IloObjective`**`(IloObjectiveI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public` **`IloObjective`**`(const IloEnv env, IloNum constant=0.0, IloObjective::Sense sense=Minimize, const char * name=0)`

This constructor creates an objective consisting of a `constant` and belonging to `env`. The sense of the objective (whether it is a minimization or maximization) is specified by `sense`; by default, it is a minimization. You may supply a `name` for the objective; by default, its `name` is the empty string. This constructor is useful when you want to create an empty objective and fill it later by column-wise modeling.

`public **IloObjective**(const IloEnv env, const IloNumExprArg expr, IloObjective::Sense sense=Minimize, const char * name=0)`

This constructor creates an objective to add to a model from `expr`.

After you create an objective from an expression with this constructor, you must use the member function `add` explicitly to add your objective to your model or to a group in order for the objective to be taken into account.

> **Note**
>
> When it accepts an expression as an argument, this constructor checks the setting of IloEnv::setNormalizer to determine whether to assume the expression has already been reduced or to reduce the expression before using it.

Methods

`public IloNum **getConstant**() const`

This member function returns the constant term from the expression of the invoking objective.

`public IloNumExprArg **getExpr**() const`

This member function returns the expression of the invoking `IloObjective` object.

`public IloObjectiveI * **getImpl**() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public IloObjective::Sense **getSense**() const`

This member function returns the sense of the invoking objective, specifying whether the objective is a minimization (`Minimize`) or a maximization (`Maximize`).

`public IloAddValueToObj **operator()**(IloNum value)`

This casting operator uses a floating-point `value` to create an instance of `IloAddNumVar` or one of its subclasses and to add that `value` to that instance. See the concept Column-Wise Modeling for an explanation of how to use this operator in column-wise modeling.

`public IloAddValueToObj **operator()**()`

This casting operator uses a floating-point `value` to create an instance of `IloAddNumVar` or one of its subclasses and to add that `value` to that instance. If no argument is given, it assumes 1.0. See the concept Column-Wise Modeling for an explanation of how to use this operator in column-wise modeling.

`public void **setConstant**(IloNum constant)`

This member function sets `constant` as the constant term in the invoking objective, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

> **Note**
>
> The member function `setConstant` notifies Concert Technology algorithms about this change of this invoking object.

```
public void setExpr(const IloNumExprArg)
```

This member function sets the expression of the invoking `IloObjective` object.

> **Note**
>
> The member function `setExpr` notifies Concert Technology algorithms about this change of this invoking object.

```
public void setLinearCoef(const IloNumVar var, IloNum value)
```

This member function sets `value` as the linear coefficient of the variable `var` in the invoking objective, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

> **Note**
>
> The member function `setLinearCoef` notifies Concert Technology algorithms about this change of this invoking object.

If you attempt to use `setLinearCoef` on a nonlinear expression, it will throw an exception on platforms that support C++ exceptions when exceptions are enabled.

```
public void setLinearCoefs(const IloNumVarArray vars, const IloNumArray values)
```

For each of the variables in `vars`, this member function sets the corresponding value of `values` (whether integer or floating-point) as its linear coefficient in the invoking objective, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

> **Note**
>
> The member function `setLinearCoefs` notifies Concert Technology algorithms about this change of this invoking object.

If you attempt to use `setLinearCoef` on a non linear expression, Concert Technology will throw an exception on platforms that support C++ exceptions when exceptions are enabled.

```
public void setSense(IloObjective::Sense sense)
```

This member function sets `sense` to specify whether the invoking objective is a maximization (`Maximize`) or minimization (`Minimize`), and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

> **Note**
>
> The member function `setSense` notifies Concert Technology algorithms about this change of this invoking object.

Inner Enumerations

Enumeration Sense

Definition file: ilconcert/ilolinear.h

Specifies objective as minimization or maximization.
An instance of the class `IloObjective` represents an objective in a model. This nested enumeration is limited in scope to that class, and its values specify the sense of an objective; that is, whether it is a minimization (`Minimize`) or a maximization (`Maximize`).

See Also: IloObjective

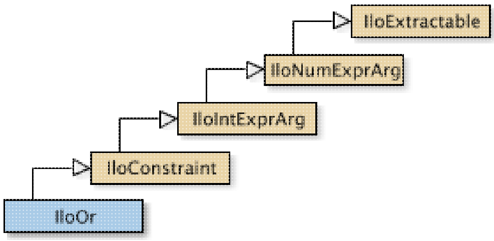
Fields:

```
Minimize = 1

Maximize = -1
```

Class IloOr

Definition file: ilconcert/ilomodel.h



Represents a disjunctive constraint.
An instance of `IloOr` represents a disjunctive constraint. In other words, it defines a disjunctive-OR among any number of constraints. Since an instance of `IloOr` is a constraint itself, you can build up extensive disjunctions by adding constraints to an instance of `IloOr` by means of the member function `IloOr::add`. You can also remove constraints from an instance of `IloOr` by means of the member function `IloOr::remove`.

The elements of a disjunctive constraint must be in the same environment.

In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Disjunctive Goals

If you would like to represent a disjunctive-OR as a goal (rather than a constraint), then you should consider the function `IloOrGoal`, documented in the *IBM ILOG Solver Reference Manual*.

What Is Extracted

All the constraints (that is, instances of `IloConstraint` or one of its subclasses) that have been added to a disjunctive constraint (an instance of `IloOr`) and that have not been removed from it will be extracted when an algorithm such as `IloCplex`, `IloCP`, or `IloSolver` extracts the constraint.

Example

For example, you may write:

```
IloOr myor(env);
myor.add(constraint1);
myor.add(constraint2);
myor.add(constraint3);
```

Those lines are equivalent to :

```
IloOr or = constraint1 || constraint2 || constraint3;
```

See Also: IloAnd, IloConstraint, operator||

Constructor Summary	
public	IloOr()
public	IloOr(IloOrI * impl)

public	IloOr(const IloEnv env, const char * name=0)

Method Summary	
public void	add(const IloConstraintArray cons) const
public void	add(const IloConstraint con) const
public IloOrI *	getImpl() const
public void	remove(const IloConstraintArray cons) const
public void	remove(const IloConstraint con) const

Inherited Methods from `IloConstraint`
getImpl

Inherited Methods from `IloIntExprArg`
getImpl

Inherited Methods from `IloNumExprArg`
getImpl

Inherited Methods from `IloExtractable`
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Constructors

public **IloOr**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IloOr**(IloOrI * impl)

This constructor creates a handle object from a pointer to an implementation object.

public **IloOr**(const IloEnv env, const char * name=0)

This constructor creates a disjunctive constraint for use in `env`. The optional argument `name` is set to `0` by default.

Methods

public void **add**(const IloConstraintArray cons) const

This member function makes all the elements in `array` elements of the invoking disjunctive constraint. In other words, it applies the invoking disjunctive constraint to all the elements of `array`.

> **Note**
>
> The member function `add` notifies Concert Technology algorithms about this change of this invoking object.

`public void add(const IloConstraint con) const`

This member function makes `constraint` one of the elements of the invoking disjunctive constraint. In other words, it applies the invoking disjunctive constraint to `constraint`.

Note

The member function `add` notifies Concert Technology algorithms about this change of this invoking object.

`public IloOrI * getImpl() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public void remove(const IloConstraintArray cons) const`

This member function removes all the elements of `array` from the invoking disjunctive constraint so that the invoking disjunctive constraint no longer applies to any of those elements.

Note

The member function `remove` notifies Concert Technology algorithms about this change of this invoking object.

`public void remove(const IloConstraint con) const`

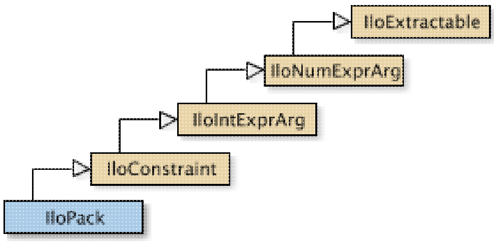
This member function removes `constraint` from the invoking disjunctive constraint so that the invoking disjunctive constraint no longer applies to `constraint`.

Note

The member function `remove` notifies Concert Technology algorithms about this change of this invoking object.

Class IloPack

Definition file: ilconcert/ilomodel.h



For constraint programming: maintains the load of containers, given weighted, assigned items.
The `IloPack` constraint maintains the load of a set of containers or bins, given a set of weighted items and an assignment of items to containers.

Consider that we have *n* items and *m* containers. Each item *i* has an integer weight *w[i]* and a constrained integer variable *p[i]* associated with it, specifying in which container (numbered contiguously from 0) item *i* is to be placed. No item can be split up, and so an item can go in only one container. Associated with each container *j* is an integer variable *l[j]* representing the load in that container; that is, the sum of the weights of the items which have been assigned to that container. A capacity can be set for each container placing an upper bound on this load variable. The constraint also ensures that the total sum of the loads of the containers is equal to the sum of the weights of the items being placed. Finally, the number, or indeed the set of containers used can be specified. A container is used if at least one item is placed in the container in question.

Constructor Summary	
public	IloPack()
public	IloPack(IloPackI * impl)
public	IloPack(const IloEnv env, const IloIntExprArray load, const IloIntExprArray where, const IloIntArray weight, const char * name=0)
public	IloPack(const IloEnv env, const IloIntExprArray load, const IloIntExprArray where, const IloIntArray weight, const IloIntExpr used, const char * name=0)
public	IloPack(const IloEnv env, const IloIntExprArray load, const IloIntExprArray where, const IloIntArray weight, const IloIntSetVar used, const char * name=0)

Method Summary	
public IloPackI *	getImpl() const

Inherited Methods from `IloConstraint`
getImpl

Inherited Methods from `IloIntExprArg`
getImpl

Inherited Methods from `IloNumExprArg`
getImpl

Inherited Methods from `IloExtractable`

```
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv,
getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr,
isObjective, isVariable, setName, setObject
```

Constructors

`public IloPack()`

This constructor creates an empty handle. You must initialize it before you use it.

`public IloPack(IloPackI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public IloPack(const IloEnv env, const IloIntExprArray load, const IloIntExprArray where, const IloIntArray weight, const char * name=0)`

This constructor creates a constraint that maintains container loads subject to the assignments of weighted items to containers.

should be placed in, counting from 0. `where` and `weight` must be the same size, otherwise an instance of `IloException` is thrown.

`public IloPack(const IloEnv env, const IloIntExprArray load, const IloIntExprArray where, const IloIntArray weight, const IloIntExpr used, const char * name=0)`

This constructor creates a constraint that maintains container loads subject to the assignments of weighted items to containers. The number of containers used is also maintained.

should be placed in, counting from 0. `where` and `weight` must be the same size, otherwise an instance of `IloException` is thrown.

`public IloPack(const IloEnv env, const IloIntExprArray load, const IloIntExprArray where, const IloIntArray weight, const IloIntSetVar used, const char * name=0)`

This constructor creates a constraint that maintains container loads subject to the assignments of weighted items to containers. The set of containers used is also maintained.

should be placed in, counting from 0. `where` and `weight` must be the same size, otherwise an instance of `IloException` is thrown. can be obtained from the set by taking the cardinality of the set.
See Also: IloCard

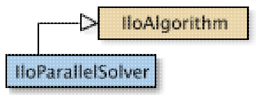
Methods

`public IloPackI * getImpl() const`

This member function returns a pointer to the implementation object of the invoking handle.

Class IloParallelSolver

Definition file: ilsolver/ilopsolver.h
Include file: <ilsolver/ilosolver.h>



`IloParallelSolver` creates multiple instances of the search engine, one in each thread, sharing the same search tree. Each of these instances is an instance of `IloSolver` and the instances are known as workers or agents. They communicate with each other across a virtual communication layer to carry out load balancing, propagation of bounds to reduce the search tree, and detection of termination (either because a solution has been found or because there is no further search to do).

See Also: IloSolver

Constructor Summary	
public	`IloParallelSolver(IloParallelSolverI * impl=0)`
public	`IloParallelSolver(const IloEnv & env, IloInt numberOfWorkers, IloInt=-1, IloBool trace=IlcFalse)`
public	`IloParallelSolver(const IloModel & model, IloInt numberOfWorkers, IloInt=-1, IloBool trace=IlcFalse)`

Method Summary	
public IloBool	`concurrentSolve(IloArray< IloGoal > goals, IloBool wait, IloGoal last=0)`
public IloParallelSolverI *	`getImpl() const`
public IloInt	`getSuccessfulWorkerId() const`
public IloSolver	`getWorker(IloInt id) const`
public void	`setFileNodeOptions(IlcInt maxSize, char * prefixName, IlcBool useCompression, IlcBool useDisk) const`
public void	`setTimeLimit(IloNum limit) const`
public void	`setTrace(IloBool trace) const`
public IloBool	`solve(IloGoal goal) const`
public IloBool	`solve() const`
public void	`unsetLimit() const`

Inherited Methods from `IloAlgorithm`
`clear, end, error, extract, getEnv, getIntValue, getIntValues, getModel, getObjValue, getStatus, getTime, getValue, getValue, getValue, getValue, getValues, getValues, isExtracted, out, printTime, resetTime, setError, setOut, setWarning, solve, warning`

Constructors

`public` **`IloParallelSolver`**`(IloParallelSolverI * impl=0)`

This constructor creates an algorithm for IBM® ILOG® Solver.

public **IloParallelSolver**(const IloEnv & env, IloInt numberOfWorkers, IloInt=-1, IloBool trace=IlcFalse)

This constructor creates an algorithm for IBM ILOG Solver. The parameter `numberOfWorkers` is used to configure load balancing.

public **IloParallelSolver**(const IloModel & model, IloInt numberOfWorkers, IloInt=-1, IloBool trace=IlcFalse)

This constructor creates an algorithm for IBM ILOG Solver. The parameter `numberOfWorkers` is used to configure load balancing.

Methods

public IloBool **concurrentSolve**(IloArray< IloGoal > goals, IloBool wait, IloGoal last=0)

This member function solves a problem by using the array of goals `goal` passed as a parameter. It solves the same model using different goals. The number of goals must be greater than or equal to the number of workers. A number of workers *n* will use the first *n* elements of the array of goals.

If there is an objective, there is cooperation among the workers. When a worker finds a new feasible solution, the bound information is propagated to the other workers. If there is no objective, there is no cooperation among the workers.

If the parameter `wait` is `IloTrue`, `concurrentSolve` will wait for all workers to finish before suceeding and returning `IloTrue`. If the parameter `wait` is `IloFalse`, `concurrentSolve` will succeed and return `IloTrue` when the first worker finishes.

The third parameter `last` is an `IloGoal` that is called by the first worker to finish. This last goal is executed after exiting from the concurrent part of the search. The parameter `last` is optional.

public IloParallelSolverI * **getImpl**() const

This member function returns a pointer to the implementation object corresponding to the invoking parallel solver.

public IloInt **getSuccessfulWorkerId**() const

This member function returns a positive integer identifying a successful worker (an instance of `IloInt`). These identifying numbers start from `0` (zero) and are contiguous. This function can only be used after a successful `IloParallelSolver::solve`.

public IloSolver **getWorker**(IloInt id) const

This member function returns the solver of the worker by identifying number.

public void **setFileNodeOptions**(IlcInt maxSize, char * prefixName, IlcBool useCompression, IlcBool useDisk) const

Node files make it possible for you to limit the amount of memory Solver uses to store open search nodes. (Open search nodes in the search are the ones which have not yet been completely explored.) You activate node files by invoking this member function. This member function sets the options for node files and must be used before starting a search.

When the memory used to store nodes is greater than `maxSize` bytes, Solver creates a buffer of one megabyte. Solver then fills that buffer with open nodes. The parameter `maxSize` cannot be less than 5 000 000. If the given parameter is less than 5 000 000, Solver will silently change it to 5 000 000.

If the parameter `useCompression` is set to `IlcTrue`, the buffer is compressed. If the parameter `useDisk` is set to `IlcTrue`, this temporary buffer is then flushed from memory and written to disk as a file. The name of the file is prefixed by `prefixName`.

If one buffer is not enough to reduce the memory consumption below `maxSize`, then Solver creates node files until the memory consumption fits that limit.

`public void` **`setTimeLimit`**`(IloNum limit) const`

This member function sets a limit on the amount of time spent during a search by `IloParallelSolver::solve`.

The limit is set to the current time plus `time`. The limit is recomputed whenever this member function is called. When the limit is reached, the search stops and the current call to the member function `IloParallelSolver` returns `IlcFalse`.

On most platforms, the time is measured in elapsed cpu seconds for the search process; on personal computers with Windows, the time is merely elapsed wall-clock time.

This change will influence any subsequent calls of the member function `IloParallelSolver::solve`.

`public void` **`setTrace`**`(IloBool trace) const`

This member function takes a Boolean value as its argument. If this argument is `IlcTrue`, it activates the trace functions of Parallel Solver. If the argument is `IlcFalse`, it inhibits the trace functions.

`public IloBool` **`solve`**`(IloGoal goal) const`

This member function solves a problem by using the `goal` passed as a parameter.

The synchronize mode is fixed to be `IloSynchronizeAndRestart`.

`public IloBool` **`solve`**`() const`

This member function solves a problem by using a default goal to launch the search.

This member function first checks to see whether a model has already been extracted. If a model has already been extracted, it then checks whether the model has changed since it was extracted previously. If the model has changed, then the model is extracted again. If the model has not changed, then it is not extracted again. In other words, this member function synchronizes with the current model before it starts its search.

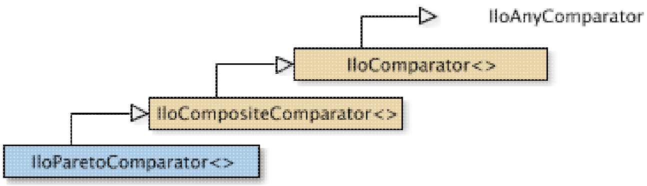
When Concert Technology determines at extraction time that the model is infeasible, it skips the remainder of the extraction, and the first call to the member function `solve` will report *without any search* that there is no solution.

```
public void unsetLimit() const
```

This member function removes any limits, such as a time limit, a limit on the number of choice points, or a limit on the number of failures, for the invoking solver. This change will influence any subsequent calls of the member functions `IloParallelSolver::solve`.

Class IloParetoComparator<>

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>



This class performs Pareto comparison of objects.
In a Pareto comparison, an object *a* is deemed better than an object *b* if and only if for all subordinate comparators *a* is never worse that *b* and *a* is strictly better than *b* in at least one comparison.

Such a comparator is useful when solutions are compared on multiple critera which have equal status.

The function `IloCompositeComparator::add` is available to you to build up your comparator by addition rather than specifying all comparators at construction time.

For more information, see Selectors.

See Also: IloComposePareto

Constructor Summary	
`public`	`IloParetoComparator(IloMemoryManager manager)` Initializes an empty Pareto composite comparator.

Inherited Methods from `IloCompositeComparator`
`add`

Inherited Methods from `IloComparator`
`isBetterOrEqual, isBetterThan, isEqual, isWorseOrEqual, isWorseThan, makeInverse, operator()`

Constructors

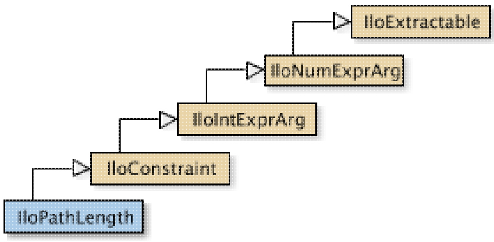
`public` **`IloParetoComparator`**`(IloMemoryManager manager)`

Initializes an empty Pareto composite comparator.

This constructor intializes an empty Pareto comparator allocated on the memory manager `manager`.

Class IloPathLength

Definition file: ilconcert/ilomodel.h



For IBM® ILOG® Solver: a constraint on accumulations along a path.
An instance of this class is a path constraint in Concert Technology.

In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

What IloPathLength Does Not Do

The path-length constraint does not determine whether there is a path between nodes in a graph; rather, it constrains accumulations (such as flow) along a path. The filtering algorithm associated with this constraint works on the accumulation variables in the array `lengths`.

If you are looking for a Hamiltonian path, for example, (that is, one in which each node is visited exactly once), consider using instead the constraint `IloAllDiff` on the variables in the array `next`.

What IloPathLength Does

If we are given

- a set of n nodes, known as N,
- a maximum number of paths among those nodes, `maxNbPaths`,
- a set of `maxNbPaths` nodes, known as S, for starting nodes,
- a set of `maxNbPaths` nodes, known as E, for ending nodes,

then a path constraint insures that there exist at most `maxNbPaths` paths starting from a node in S, visiting nodes in N, and ending at a node in E. Furthermore, each node will be *visited* only once, has only one predecessor and only one successor, and each node *belongs* to a path that starts from a node in S and ends at a node in E.

In particular, in an instance of `IloPath`, in the arrays `next` and `cumul`,

- the indices in `[0, n-1]` correspond to the nodes of N,
- the indices in `[n, n+maxNbPaths-1]` correspond to the nodes of E,
- and the indices in `[n+maxNbPaths, n+2*maxNbPaths-1]` correspond to the nodes of S.

In other words, the size of `next` and `cumul` is `n+2*maxNbPaths`.

`next[i]` is the node following node `i` on the current path. `cumul[i]` is the accumulated cost from the beginning of the path to node `i`. The argument `transit` specifies the *transition function*.

When this constraint is satisfied, it insures that for all indices `i` in the range `[0, n-1]` or in `[n+maxNbPaths, n+2*maxNbPaths-1]`, if `next[i]==j` and `j` is in `[0, n+maxNbPaths-1]`, then `cumul[i] + transit.transit(i,j) <= cumul[j]`.

When `i` is in the range `[n, n+maxNbPaths-1]`, `next[i]` has no meaning because the nodes in E do not have successors, of course. In this case, the constraint deals with them by setting `next[i]` to `i+maxNbPaths` (that is, nodes of S).

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloAllDiff, IloConstraint, IloPathTransitFunction, IloPathTransitI

Constructor Summary	
public	`IloPathLength()`
public	`IloPathLength(IloPathLengthI * impl)`
public	`IloPathLength(const IloEnv env, const IloIntVarArray next, const IloNumVarArray cumul, IloPathTransitFunction transit, IloInt nbPaths=1, const char * name=0)`
public	`IloPathLength(const IloEnv env, const IloIntVarArray next, const IloIntVarArray cumul, IloPathTransitFunction transit, IloInt nbPaths=1, const char * name=0)`
public	`IloPathLength(const IloEnv env, const IloIntVarArray next, const IloNumVarArray cumul, IloPathTransitI * pathTransit, IloInt nbPaths=1, const char * name=0)`
public	`IloPathLength(const IloEnv env, const IloIntVarArray next, const IloIntVarArray cumul, IloPathTransitI * pathTransit, IloInt nbPaths=1, const char * name=0)`

Method Summary	
`public IloPathLengthI *`	`getImpl() const`

Inherited Methods from `IloConstraint`
`getImpl`

Inherited Methods from `IloIntExprArg`
`getImpl`

Inherited Methods from `IloNumExprArg`
`getImpl`

Inherited Methods from `IloExtractable`
`asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject`

Constructors

`public` **`IloPathLength`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IloPathLength`**`(IloPathLengthI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public` **`IloPathLength`**`(const IloEnv env, const IloIntVarArray next, const`

```
IloNumVarArray cumul, IloPathTransitFunction transit, IloInt nbPaths=1, const char
* name=0)
```

This constructor creates a path constraint in an environment.

In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

If the array `next` or `cumul` is not an appropriate length, then on platforms that support C++ exceptions when exceptions are enabled, this constructor will throw the exception `InvalidArraysException`.

```
public IloPathLength(const IloEnv env, const IloIntVarArray next, const
IloIntVarArray cumul, IloPathTransitFunction transit, IloInt nbPaths=1, const char
* name=0)
```

This constructor creates a path constraint in an environment.

In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

If the array `next` or `cumul` is not an appropriate length, then on platforms that support C++ exceptions when exceptions are enabled, this constructor will throw the exception `InvalidArraysException`.

```
public IloPathLength(const IloEnv env, const IloIntVarArray next, const
IloNumVarArray cumul, IloPathTransitI * pathTransit, IloInt nbPaths=1, const char *
name=0)
```

This constructor creates a path constraint in an environment.

In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

If the array `next` or `cumul` is not an appropriate length, then on platforms that support C++ exceptions when exceptions are enabled, this constructor will throw the exception `InvalidArraysException`.

```
public IloPathLength(const IloEnv env, const IloIntVarArray next, const
IloIntVarArray cumul, IloPathTransitI * pathTransit, IloInt nbPaths=1, const char *
name=0)
```

This constructor creates a path constraint in an environment.

In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

If the array `next` or `cumul` is not an appropriate length, then on platforms that support C++ exceptions when exceptions are enabled, this constructor will throw the exception `InvalidArraysException`.

Methods

```
public IloPathLengthI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

Class IloPathTransitI

Definition file: ilconcert/ilomodel.h

For IBM® ILOG® Solver: a transit function in a path constraint.
You can define the transit function in a path constraint (the cost for linking two nodes together).

This class is the implementation class for `IloPathTransit`, the class of object that defines a transit function for the path constraint. The virtual member function `transit` in `IlcPathTransitI` returns the transition cost for connecting two nodes together.

To express new transit functions, you can define a subclass of `IlcPathTransitI`. If this transition can be expressed by an evaluation function, then you can use the predefined `IloPathFunction` for that purpose.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloPathLength, IloPathTransitFunction

Constructor and Destructor Summary	
public	IloPathTransitI()

Method Summary	
public virtual IloPathTransitI *	makeClone(IloEnvI *) const
public virtual IloNum	transit(IloInt i, IloInt j)

Constructors and Destructors

`public` **`IloPathTransitI`**`()`

This constructor creates an implementation of a transit function.

Methods

`public virtual IloPathTransitI *` **`makeClone`**`(IloEnvI *) const`

This virtual member function returns a copy of the invoking object.

`public virtual IloNum` **`transit`**`(IloInt i, IloInt j)`

This virtual member function returns the transition cost from node `i` to node `j`. Its default implementation returns 0 (zero) as the value of every transition.

Class IloPoolOperator

Definition file: ilsolver/iimoperator.h
Include file: <ilsolver/iim.h>

The pool operator class.
Pool operators are Solver related entities which take an input in the form of a solution pool and instantiate Solver's constrained variables to create a solution therein. Importantly, operators can be cast to processors (`IloPoolProc`). This cast operator results in a processor which will transform the so-instantiated solver into an instance of `IloSolution`, based on a given solution prototype.

You can define custom operators by using the `ILOIIMOP0` macro (or a variant). Numerous built-in operators are also provided, easily accessible from `IloEAOperatorFactory`.

For the most part, pool operators can be thought of as standard Solver goals with the additional property that they have access to an input pool of solutions which they use to influence their behavior.

See Also: IloEAOperatorFactory, ILOIIMOP0

Constructor Summary	
public	`IloPoolOperator(IloGoal goal)` Creates an operator from a goal.

Method Summary	
public void	`addListener(IloListener listener) const` Adds a listener to the operator.
public void	`end()`
public const char *	`getDisplayName() const` Get the display name of an operator.
public IloEnv	`getEnv() const`
public const char *	`getName() const`
public IloAny	`getObject() const`
public IloSolution	`getPrototype() const` Returns the solution prototype.
public	`operator IloPoolProc() const` Casts a pool operator to a pool processor.
public IloPoolProc	`operator()(IlcInt size) const` Specify the number of solutions to produce.
public void	`removeListener(IloListener listener) const` Removes a listener from an operator.
public void	`setName(const char * name) const`
public void	`setObject(IloAny obj) const`

public void	`setPrototype(IloSolution prototype) const` Sets the solution prototype.

Inner Class	
IloPoolOperator::Event	Event produced by pool operators.
IloPoolOperator::InvocationEvent	The event produced by pool operators when they are invoked.
IloPoolOperator::SuccessEvent	The event produced by pool operators when they are involved in the production of a solution.

Constructors

`public` **`IloPoolOperator`**`(IloGoal goal)`

Creates an operator from a goal.

This constructor creates an operator from a goal. When the created operator is executed, it will behave exactly as `goal`. Most often this constructor is useful when an operator needs to be combined with a goal using `goal && op` or `op && goal`. The constructor makes sure that a goal can be cast to an operator so that the `&&` operator on pool operators can then be applied.

See Also: operator &&(IloPoolOperator, IloPoolOperator)

Methods

`public void` **`addListener`**`(IloListener listener) const`

Adds a listener to the operator.

This member function adds a listener to the operator. Depending on the type of listener added to the operator, based on either `IloPoolOperator::InvocationEvent` or `IloPoolOperator::SuccessEvent`, the listener will be activated when the operator is invoked, or when the operator participates in the production of a solution.

See Also: IloListener, ILOIIMLISTENER0, IloPoolOperator::SuccessEvent, IloPoolOperator::InvocationEvent

`public void` **`end`**`()`

brief Destroys the object.

This function deletes the object from the environment on which it was allocated and sets the implementation pointer to zero.

`public const char *` **`getDisplayName`**`() const`

Get the display name of an operator.

This member function returns the display name of the operator, which is useful for display purposes when type information is required on the operator. The name returned will be based on the function of the operator.

`public IloEnv` **`getEnv`**`() const`

brief Returns the allocation environment.

This function returns the environment on which the invoking object was allocated.

`public const char * **getName**() const`

This member function returns a character string specifying the name of the invoking object (if there is one).

`public IloAny **getObject**() const`

This member function returns the object associated with the invoking object (if there is one). Normally, an associated object contains user data pertinent to the invoking object.

`public IloSolution **getPrototype**() const`

Returns the solution prototype.

This member function returns the solution prototype set in the previous call to `IloPoolOperator::setPrototype` or an empty handle if none was set.

`public **operator IloPoolProc**() const`

Casts a pool operator to a pool processor.

This operator casts a pool operator to a pool processor. Objects of type `IloPoolOperator` can do very little work on their own; it is only in casting one to a *processor* that more useful work can be done with it.

When the cast operator is invoked, a pool processor is created which performs the following when invoked:

- Asks for input and supplies it to the operator.
- Invokes the operator.
- Searches for a solution prototype, either in the operator, or, failing this, in the input pool of the processor.
- Clones the prototype, saves it, and adds it to the output pool of the processor.

The resulting pool processor can be used with selectors and other processors via `>>` to create chains.

`public IloPoolProc **operator()**(IlcInt size) const`

Specify the number of solutions to produce.

This operator specifies the number of solutions to produce, which will be at least `size` solutions. The code `op(n)` is shorthand for `IloPoolProc(op)(n)` where `op` is of type `IloPoolOperator` and `n` is an integer.

`public void **removeListener**(IloListener listener) const`

Removes a listener from an operator.

This member function removes a previously added listener from an operator. After the removal, `listener` will no longer be called when the operator is invoked or succeeds.

`public void **setName**(const char * name) const`

This member function assigns `name` to the invoking object.

`public void **setObject**(IloAny obj) const`

This member function associates `obj` with the invoking object. The member function `getObject` accesses this associated object afterward. Normally, `obj` contains user data pertinent to the invoking object.

`public void **setPrototype**(IloSolution prototype) const`

Sets the solution prototype.

This member function sets the solution prototype. It indicates to an operator how the solution produced in constrained variables should be saved. When a solution needs to be created, the prototype is cloned and the Solver variables specified in the clone are saved using the `IloSolution::store`.

Setting the prototype in this manner is not normally necessary, unless quite specialized behavior is required for a particular operator. Normally, the prototype can be set more simply on an operator factory, ensuring that all operators produced by the factory will have the correct prototype. Even if this is not done, a prototype can be found by examination of the input pool of the processor which calls the operator.

Class IloPoolOperatorFactory

Definition file: ilsolver/iimoperator.h
Include file: <ilsolver/iim.h>

An operator factory class providing services for simplifying operator creation.
This class eases `IloPoolOperator` configuration. It allows you to specify common operator parameters at one time before creating many operators. Objects of this class are usually used through one if its subclasses.

See Also: IloPoolOperator, IloListener, IloSearchLimit

Constructor Summary	
`public`	`IloPoolOperatorFactory(IloEnv env)` Creates a factory that can be used to configure pool operators.

Method Summary	
`public void`	`addAfterOperate(IloPoolOperator op) const` Adds a final operator to execute.
`public void`	`addBeforeOperate(IloPoolOperator op) const` Adds an initial operator to execute.
`public void`	`addListener(IloListener listener) const` Adds a listener to the factory.
`public void`	`end()`
`public IloPoolOperator`	`getAfterOperate() const` Returns the final operator to execute.
`public IloPoolOperator`	`getBeforeOperate() const` Returns the initial operator to execute.
`public IloEnv`	`getEnv() const`
`public const char *`	`getName() const`
`public IloAny`	`getObject() const`
`public IloSolution`	`getPrototype() const` Returns the solution prototype.
`public IloSearchLimit`	`getSearchLimit() const` Returns the search limit.
`public IloPoolOperator`	`operator()(IloPoolOperator op, const char * name=0) const` Configures the given pool operator using the factory parameters.
`public void`	`removeListener(IloListener listener) const` Removes a listener previously added to the factory.

public void	`setAfterOperate(IloPoolOperator op) const` Sets a final operator to execute.
public void	`setBeforeOperate(IloPoolOperator op) const` Sets an initial operator to execute.
public void	`setName(const char * name) const`
public void	`setObject(IloAny obj) const`
public void	`setPrototype(IloSolution prototype) const` Sets the solution prototype used by created operators.
public void	`setSearchLimit(IloSearchLimit searchLimit) const` Sets a search limit on operators.

Constructors

`public` **`IloPoolOperatorFactory`**`(IloEnv env)`

Creates a factory that can be used to configure pool operators.

This constructor creates a factory that can be used to configure pool operators. If you wish to create a factory which build *evolutionary* operators, use the subclass `IloEAOperatorFactory`.

Methods

`public void` **`addAfterOperate`**`(IloPoolOperator op) const`

Adds a final operator to execute.

This member function adds a final operator to execute. Any operator of the factory will have operator `op` executed after it. Note that unlike `IloPoolOperatorFactory::setAfterOperate`, other operators already set will not be replaced. Also note that `op` will only be executed if the factory operator executed successfully.

See Also: operator &&(IloPoolOperator, IloPoolOperator) , IloPoolOperator::IloPoolOperator

`public void` **`addBeforeOperate`**`(IloPoolOperator op) const`

Adds an initial operator to execute.

This member function adds an initial operator to execute. Any operator of the factory will have operator `op` executed before it. Note that unlike `IloPoolOperatorFactory::setBeforeOperate`, other operators already added will not be replaced.

Note Note the ordering of execution of "before" operators. If you perform `factory.addBeforeOperate(a)` followed by `factory.addBeforeOperate(b)`, then any operator created by the factory will execute `b`, then `a`, then the operator in question.

See Also: operator &&(IloPoolOperator, IloPoolOperator)

`public void` **`addListener`**`(IloListener listener) const`

Adds a listener to the factory.

This member function adds a listener to the factory. Any operator then created by the factory will have the specified listener added.

See Also: IloPoolOperator::addListener

`public void` **`end`**`()`

brief Destroys the object.

This function deletes the object from the environment on which it was allocated and sets the implementation pointer to zero.

`public IloPoolOperator` **`getAfterOperate`**`() const`

Returns the final operator to execute.

This member function returns the final operator to execute, which was last set using `IloPoolOperatorFactory#setAfterOperator`, or an empty handle if no "after" operator was set.

`public IloPoolOperator` **`getBeforeOperate`**`() const`

Returns the initial operator to execute.

This member function returns the initial operator to execute, which was last set using `setBeforeOperator`, or an empty handle if no "before" operator was set.

`public IloEnv` **`getEnv`**`() const`

brief Returns the allocation environment.

This function returns the environment on which the invoking object was allocated.

`public const char *` **`getName`**`() const`

This member function returns a character string specifying the name of the invoking object (if there is one).

`public IloAny` **`getObject`**`() const`

This member function returns the object associated with the invoking object (if there is one). Normally, an associated object contains user data pertinent to the invoking object.

`public IloSolution` **`getPrototype`**`() const`

Returns the solution prototype.

This member function returns the solution prototype last set using `IloPoolOperatorFactory::setPrototype` or an empty handle if none has been set.

`public IloSearchLimit` **`getSearchLimit`**`() const`

Returns the search limit.

This member function returns the search limit last set using `IloPoolOperatorFactory::setSearchLimit` or an empty handle if none was set.

```
public IloPoolOperator operator()(IloPoolOperator op, const char * name=0) const
```

Configures the given pool operator using the factory parameters.

This operator configures the given pool operator using the factory parameters. A new operator will be produced using pool operator `op` as a template, but which will be parameterized via the parameters of the factory. That is, it will have a prototype, limit, and listeners added as specified by the factory.

See Also: IloPoolOperator::addListener, IloPoolOperator::setPrototype, IloPoolOperatorFactory::setSearchLimit, IloPoolOperator::IloPoolOperator

```
public void removeListener(IloListener listener) const
```

Removes a listener previously added to the factory.

This member function removes a listener added to the factory. Any operator then created by the factory will no longer have the specified listener added.

See Also: IloPoolOperator::removeListener

```
public void setAfterOperate(IloPoolOperator op) const
```

Sets a final operator to execute.

This member function sets a final operator to execute. Any operator of the factory will have operator `op` executed after it. Note that `op` will only be executed if the factory operator executed successfully.

See Also: operator &&(IloPoolOperator, IloPoolOperator)

```
public void setBeforeOperate(IloPoolOperator op) const
```

Sets an initial operator to execute.

This member function sets an initial operator to execute. Any operator then created by the factory will have the "before" operator `op` executed before it.

See Also: operator &&(IloPoolOperator, IloPoolOperator)

```
public void setName(const char * name) const
```

This member function assigns `name` to the invoking object.

```
public void setObject(IloAny obj) const
```

This member function associates `obj` with the invoking object. The member function `getObject` accesses this associated object afterward. Normally, `obj` contains user data pertinent to the invoking object.

```
public void setPrototype(IloSolution prototype) const
```

Sets the solution prototype used by created operators.

This member function sets the solution prototype used by created operators. Any operator created by the factory will then have its prototype set accordingly.

See Also: IloPoolOperator::setPrototype

```
public void setSearchLimit(IloSearchLimit searchLimit) const
```

Sets a search limit on operators.

This member functions sets a search limit on operators. This search limit will apply to any operator then created by the factory.

Class IloPoolProc

Definition file: ilsolver/iimiloproc.h
Include file: <ilsolver/iim.h>

Pool processor.
An `IloPoolProc` is an object dedicated to solution pool processing. Such pool processors can be chained together using the `>>` operator of the C++ language. This chaining specifies how pool elements flow between processors. For example, the following code solves a goal composed by chaining solution processors:

```
IloSolutionPool initialPool = ...;
IloPoolProc producer = ...;
IloPoolProc consumer = ...;
IloSolutionPool resultPool = ...;
solver.solve(IloExecuteProcessor(
  initialPool >> producer(10) >> consumer >> resultPool
));
```

The `>>` operator is overloaded so that:

- Pool processors are created for each pool (namely `initialPool` and `resultPool`).
- All the processors are chained so that information can be transferred between processors.

The `()` operator is overloaded and used to specify how many pool elements should be generated by a processor. In the above example, the code specifies that `producer` should generate ten solutions.

Constructor Summary	
`public`	`IloPoolProc(IloSolutionPool pool)` Creates a pool processor from a solution pool.

Method Summary	
`public void`	`end()`
`public const char *`	`getDisplayName() const` Get the "display name" of a processor.
`public IloEnv`	`getEnv() const`
`public const char *`	`getName() const`
`public IloAny`	`getObject() const`
`public IloPoolProc`	`operator()(IlcInt size) const` Specifies the desired number of outputs to a processor.
`public void`	`setName(const char * name) const`
`public void`	`setObject(IloAny obj) const`

Constructors

`public` **`IloPoolProc`**`(IloSolutionPool pool)`

Creates a pool processor from a solution pool.

This constructor creates a processor from a solution pool. The created processor acts as a type of synchronizer such that the solutions transferred on the resulting processor's output port will also be placed in the solution pool `pool` passed as argument.

The behavior of the created processor will depend upon its environment, and in particular, if the processor is supplied an input using a >> to its left. If the processor is supplied an input, then the processor places that input into `pool`, while also transferring to the output. If, on the other hand, the processor is not supplied any input, then the contents of `pool` are placed on the output.

Assuming you have "population" and "offspring" pools as well as a processor "proc" able to process the population to produce offspring, you can perform a processing chain such as:

```
IloPoolProc chain = population >> proc >> offspring;
```

where the `population` pool feeds the processor `proc`, whose result is stored in the `offspring` pool.

> **Note**
>
> It is often convenient to insert a pool in a processing chain for debugging purposes. For monitoring pool dynamics, you can add listeners to the pool to be notified when its state changes.

See Also: operator>>

Methods

`public void` **`end`**`()`

brief Destroys the object.

This function deletes the object from the environment on which it was allocated and sets the implementation pointer to zero.

`public const char *` **`getDisplayName`**`() const`

Get the "display name" of a processor.

This member function returns the display name of the processor, which is useful for display purposes when type information is required on the processor. The name returned will be based on the function of the processor.

`public IloEnv` **`getEnv`**`() const`

brief Returns the allocation environment.

This function returns the environment on which the invoking object was allocated.

`public const char *` **`getName`**`() const`

This member function returns a character string specifying the name of the invoking object (if there is one).

`public IloAny` **`getObject`**`() const`

This member function returns the object associated with the invoking object (if there is one). Normally, an associated object contains user data pertinent to the invoking object.

`public IloPoolProc` **`operator()`**`(IlcInt size) const`

Specifies the desired number of outputs to a processor.

This operator creates a new pool processor that wraps the invoked pool processor and retrieves the number of elements specified by `size`. This operator is used to specify the number of outputs to a processor.

The following code initializes a population with `populationSize` solutions. In this particular case, the `()` operator is applied to the processor returned by the `randomize` method of the `IloEAOperatorFactory` factory:

```
solver.solve(IloExecuteProcessor(
  env, factory.randomize("rand")(populationSize) >> population
));
```

`public void` **`setName`**`(const char * name) const`

This member function assigns `name` to the invoking object.

`public void` **`setObject`**`(IloAny obj) const`

This member function associates `obj` with the invoking object. The member function `getObject` accesses this associated object afterward. Normally, `obj` contains user data pertinent to the invoking object.

Class IloPredicate<>

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

The `IloPredicate` template class is the base class for all predicates. A predicate is a class that implements a test on an object (using `operator()`), and returns an `IloBool` value. Such classes can be used (along with classes of `IloComparator`) to parameterize selectors used in goal-based search.

Predicates can easily be built by using the `ILOPREDICATE0` macros or by composing existing predicates and evaluators.

For more information, see Selectors.

See Also: IloComparator, IloEvaluator, IloTranslator

Method Summary	
`public IloBool`	`operator()(IloObject o, IloAny nu=0) const`

Methods

`public IloBool` **`operator()`**`(IloObject o, IloAny nu=0) const`

This operator implements the test of an instance of class `IloObject` that may use a user-given context `nu`. If the predicate is a composite predicate (see operators on predicates), the context is also passed to components (predicates and evaluators) involved in the computation of the composite predicate.

Class IloRandom

Definition file: ilconcert/ilorandom.h

This handle class produces streams of pseudo-random numbers.
You can use objects of this class to create a search with a random element. You can create any number of instances of this class; these instances insure reproducible results in multithreaded applications, where the use of a single source for random numbers creates problems.

See Also the class `IloRandomize` in the *IBM ILOG Solver Reference Manual*.

Constructor Summary	
public	IloRandom()
public	IloRandom(const IloEnv env, IloInt seed=0)
public	IloRandom(IloRandomI * impl)
public	IloRandom(const IloRandom & rand)

Method Summary	
public void	end()
public IloEnv	getEnv() const
public IloNum	getFloat() const
public IloRandomI *	getImpl() const
public IloInt	getInt(IloInt n) const
public const char *	getName() const
public IloAny	getObject() const
public void	reSeed(IloInt seed)
public void	setName(const char * name) const
public void	setObject(IloAny obj) const

Constructors

public **IloRandom**()

This constructor creates a random number generator; it is initially an empty handle. You must assign this handle before you use its member functions.

public **IloRandom**(const IloEnv env, IloInt seed=0)

This constructor creates an object that generates random numbers. You can seed the generator by supplying a value for the integer argument `seed`.

public **IloRandom**(IloRandomI * impl)

This constructor creates a handle object (an instance of the class `IloRandom`) from a pointer to an implementation object (an instance of the class `IloRandomI`).

```
public IloRandom(const IloRandom & rand)
```

This constructor creates a handle object from a reference to a random number generator. After execution, both the newly constructed handle and `rand` point to the same implementation object.

Methods

```
public void end()
```

This member function releases all memory used by the random number generator. After a call to this member function, you should not use the generator again.

```
public IloEnv getEnv() const
```

This member function returns the environment associated with the implementation class of the invoking generator.

```
public IloNum getFloat() const
```

This member function returns a floating-point number drawn uniformly from the interval `[0..1)`.

```
public IloRandomI * getImpl() const
```

This member function returns the implementation object of the invoking handle.

```
public IloInt getInt(IloInt n) const
```

This member function returns an integer drawn uniformly from the interval `[0..n)`.

```
public const char * getName() const
```

This member function returns a character string specifying the name of the invoking object (if there is one).

```
public IloAny getObject() const
```

This member function returns the object associated with the invoking object (if there is one). Normally, an associated object contains user data pertinent to the invoking object.

```
public void reSeed(IloInt seed)
```

This member function re-seeds the random number generator with `seed`.

```
public void setName(const char * name) const
```

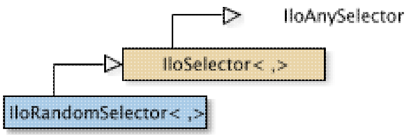
This member function assigns `name` to the invoking object.

```
public void setObject(IloAny obj) const
```

This member function associates `obj` with the invoking object. The member function `getObject` accesses this associated object afterward. Normally, `obj` contains user data pertinent to the invoking object.

Class IloRandomSelector<,>

Definition file: ilsolver/iimmulti.h
Include file: <ilsolver/iim.h>



A selector which chooses objects randomly.
This class is a selector which randomly selects an object from a given container. The selection is unbiased, with each object having an equal probability of selection.

When you create a random selector, you must pass a visitor object. This visitor will traverse the container from which you are selecting objects. However, in the case where a *default visitor* exists for the container, you can omit the visitor as the default will be used. In addition to the default visitors available in Solver, IIM provides default visitors for `IloSolutionPool` and `IloPoolProcArray`.

The following code shows how to create a random selector and use it to build a pool processor:

```
// Selects parents randomly
IloPoolProc selector = IloSelectSolutions(
  env, IloRandomSelector<IloSolution, IloSolutionPool>(env)
);
```

> **Note**
>
> An instance of `IloRandomSelector` which has been transformed into a pool processor using `IloSelectSolutions` will always draw its random numbers from the random number generator of the solver on which it is executing.

See Also: IloSolutionPool, IloPoolProcArray, IloVisitor, IloEvaluator, IloTournamentSelector, IloRouletteWheelSelector

Constructor Summary	
`public`	`IloRandomSelector(IloEnv env, IloVisitor< IloObject, IloContainer > visitor=0)` Builds a random selector.

Method Summary	
`public IloVisitor< IloObject, IloContainer >`	`getVisitor() const` Delivers the visitor used by the selector.

Inherited Methods from `IloSelector`
`select`

Constructors

`public` **`IloRandomSelector`**`(IloEnv env, IloVisitor< IloObject, IloContainer > visitor=0)`

Builds a random selector.

This constructor builds a random selector from an environment `env` and an optional visitor `visitor`. If no visitor is specified, a default visitor will be used if it exists. If no default visitor exists, an exception of type `IloException` is raised.

Methods

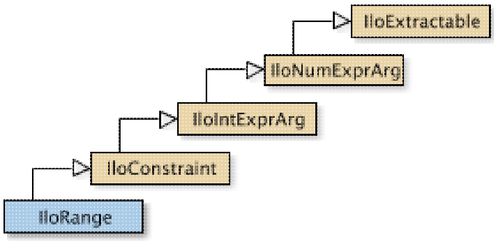
`public IloVisitor< IloObject, IloContainer >` **`getVisitor`**`() const`

Delivers the visitor used by the selector.

This member function returns the visitor passed at construction time or, if no visitor was passed, the default visitor for the container from which objects are being selected.

Class IloRange

Definition file: ilconcert/ilolinear.h



An instance of this class is a range in a model.
This class models a constraint of the form:

```
lowerBound <= expression <= upperBound
```

You can create a range from the constructors in this class or from the arithmetic operators on numeric variables (instances of `IloNumVar` and its subclasses) or on expressions (instances of `IloExpr` and its subclasses):

- `operator <=`
- `operator >=`
- `operator ==`

After you create a constraint, such as an instance of `IloRange`, you must explicitly add it to the model in order for it to be taken into account. To do so, use the member function `IloModel::add` to add the range to a model and the member function `IloAlgorithm::extract` to extract the model for an algorithm.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

What Is Extracted

All the variables (that is, instances of `IloNumVar` or one of its subclasses) in the range (an instance of `IloRange`) will be extracted when an algorithm such as `IloCplex`, documented in the *IBM ILOG CPLEX Reference Manual*, extracts the range.

Normalizing Linear Expressions: Reducing the Terms

Normalizing is sometimes known as *reducing the terms* of a linear expression.

Linear expressions consist of terms made up of constants and variables related by arithmetic operations; for example, x + 3y. In some linear expressions, a given variable may appear in more than one term, for example, x + 3y +2x. Concert Technology has more than one way of dealing with linear expressions in this respect, and you control which way Concert Technology treats linear expressions from your application.

In one mode (the default mode), Concert Technology analyzes linear expressions that your application passes it, and attempts to reduce them so that a given variable appears in only one term in the expression. You set this mode with the member function `IloEnv::setNormalizer(IloTrue)` .

Certain constructors and member functions in this class check this setting in the model and behave accordingly: they attempt to reduce expressions. This mode may require more time during preliminary computation, but it avoids the possibility of an assertion in some of the member functions of this class failing in the case of duplicates.

In the other mode, Concert Technology assumes that no variable appears in more than one term in any of the linear expressions that your application passes to Concert Technology. We call this mode assume normalized linear expressions. You set this mode with the member function `IloEnv::setNormalizer(IloFalse)`.

Certain constructors and member functions in this class check this setting in the model and behave accordingly: they assume that no variable appears in more than one term in an expression. This mode may save time during computation, but it entails the risk that an expression may contain one or more variables, each of which appears in one or more terms. This situation will cause certain `assert` statements in Concert Technology to fail if you do not compile with the flag `-DNDEBUG`.

See Also: IloConstraint, IloRangeArray

Constructor Summary	
public	IloRange()
public	IloRange(IloRangeI * impl)
public	IloRange(const IloEnv env, IloNum lb, IloNum ub, const char * name=0)
public	IloRange(const IloEnv env, IloNum lhs, const IloNumExprArg expr, IloNum rhs=IloInfinity, const char * name=0)
public	IloRange(const IloEnv env, const IloNumExprArg expr, IloNum rhs=IloInfinity, const char * name=0)

Method Summary	
public IloNumExprArg	getExpr() const
public IloRangeI *	getImpl() const
public IloNum	getLB() const
public IloNum	getUB() const
public IloAddValueToRange	operator()(IloNum value) const
public void	setBounds(IloNum lb, IloNum ub)
public void	setExpr(const IloNumExprArg expr)
public void	setLB(IloNum lb)
public void	setLinearCoef(const IloNumVar var, IloNum value)
public void	setLinearCoefs(const IloNumVarArray vars, const IloNumArray values)
public void	setUB(IloNum ub)

Inherited Methods from `IloConstraint`
getImpl

Inherited Methods from `IloIntExprArg`
getImpl

Inherited Methods from `IloNumExprArg`
getImpl

Inherited Methods from `IloExtractable`
asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv, getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr, isObjective, isVariable, setName, setObject

Constructors

`public IloRange()`

This constructor creates an empty handle. You must initialize it before you use it.

`public IloRange(IloRangeI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public IloRange(const IloEnv env, IloNum lb, IloNum ub, const char * name=0)`

This constructor creates an empty range constraint. Before you use this constraint, you must fill the range. The optional argument `name` is set to `0` by default.

After you create a range constraint, you must explicitly add it to a model in order for it to be taken into account. To do so, use the member function `IloModel::add`.

```
public IloRange(const IloEnv env, IloNum lhs, const IloNumExprArg expr, IloNum
rhs=IloInfinity, const char * name=0)
```

This constructor creates a range constraint from an expression (an instance of the class `IloNumExprArg`) and its upper bound (`rhs`). The default bound for `rhs` is the symbolic constant `IloInfinity`. The optional argument `name` is set to `0` by default.

Note

When it accepts an expression as an argument, this constructor checks the setting of `IloEnv::setNormalizer` to determine whether to assume the expression has already been reduced or to reduce the expression before using it.

```
public IloRange(const IloEnv env, const IloNumExprArg expr, IloNum rhs=IloInfinity,
const char * name=0)
```

This constructor creates a range constraint from an expression (an instance of the class `IloNumExprArg`) and its upper bound (`rhs`). Its lower bound (`lhs`) will be `-IloInfinity`. The default bound for `rhs` is `IloInfinity`. The optional argument `name` is set to `0` by default.

Note

When it accepts an expression as an argument, this constructor checks the setting of `IloEnv::setNormalizer` to determine whether to assume the expression has already been reduced or to reduce the expression before using it.

Methods

`public IloNumExprArg getExpr() const`

This member function returns the expression of the invoking `IloRange` object.

`public IloRangeI * getImpl() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public IloNum getLB() const`

This member function returns the lower bound of the invoking range.

`public IloNum getUB() const`

This member function returns the upper bound of the invoking range.

`public IloAddValueToRange operator()(IloNum value) const`

This operator creates the objects needed internally to represent a range in column-wise modeling. See the concept Column-Wise Modeling for an explanation of how to use this operator in column-wise modeling.

`public void setBounds(IloNum lb, IloNum ub)`

This member function sets `lb` as the lower bound and `ub` as the upper bound of the invoking range, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

Note

The member function `setBounds` notifies Concert Technology algorithms about this change of this invoking object.

`public void setExpr(const IloNumExprArg expr)`

This member function sets `expr` as the invoking range, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

Note

The member function `setExpr` notifies Concert Technology algorithms about this change of this invoking object.

`public void setLB(IloNum lb)`

This member function sets `lb` as the lower bound of the invoking range, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

Note

The member function `setLB` notifies Concert Technology algorithms about this change of this invoking object.

`public void setLinearCoef(const IloNumVar var, IloNum value)`

This member function sets `value` as the linear coefficient of the variable `var` in the invoking range, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

> **Note**
>
> The member function `setLinearCoef` notifies Concert Technology algorithms about this change of this invoking object.

If you attempt to use `setLinearCoef` on a non linear expression, it will throw an exception on platforms that support C++ exceptions when exceptions are enabled.

```
public void setLinearCoefs(const IloNumVarArray vars, const IloNumArray values)
```

For each of the variables in `vars`, this member function sets the corresponding value of `values` (whether integer or floating-point) as its linear coefficient in the invoking range, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

> **Note**
>
> The member function `setLinearCoefs` notifies Concert Technology algorithms about this change of this invoking object.

If you attempt to use `setLinearCoef` on a non linear expression, it will throw an exception on platforms that support C++ exceptions when exceptions are enabled.

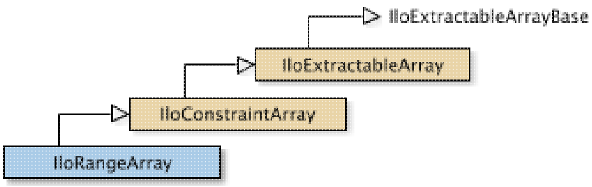
```
public void setUB(IloNum ub)
```

This member function sets `ub` as the upper bound of the invoking range, and it creates the appropriate instance of the undocumented class `IloChange` to notify algorithms about this change of an extractable object in the model.

> **Note**
>
> The member function `setUB` notifies Concert Technology algorithms about this change of this invoking object.

Class IloRangeArray

Definition file: ilconcert/ilolinear.h



The array class of ranges for a model.
For each basic type, Concert Technology defines a corresponding array class. `IloRangeArray` is the array class of ranges for a model.

Instances of `IloRangeArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added to or removed from the array.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

For information on arrays, see the concept Arrays

> **Note**
>
> `IloRangeArray` has access to member functions defined in the `IloArray` template.

See Also: IloRange, operator>>, operator<<

Constructor Summary	
public	`IloRangeArray(IloDefaultArrayI * i=0)`
public	`IloRangeArray(const IloEnv env, IloInt n=0)`
public	`IloRangeArray(const IloEnv env, IloInt n, IloNum lb, IloNum ub)`
public	`IloRangeArray(const IloEnv env, const IloNumArray lbs, const IloNumExprArray rows, const IloNumArray ubs)`
public	`IloRangeArray(const IloEnv env, IloNum lb, const IloNumExprArray rows, const IloNumArray ubs)`
public	`IloRangeArray(const IloEnv env, const IloNumArray lbs, const IloNumExprArray rows, IloNum ub)`
public	`IloRangeArray(const IloEnv env, IloNum lb, const IloNumExprArray rows, IloNum ub)`
public	`IloRangeArray(const IloEnv env, const IloIntArray lbs, const IloNumExprArray rows, const IloIntArray ubs)`
public	`IloRangeArray(const IloEnv env, IloNum lb, const IloNumExprArray rows, const IloIntArray ubs)`
public	`IloRangeArray(const IloEnv env, const IloIntArray lbs, const IloNumExprArray rows, IloNum ub)`
public	`IloRangeArray(const IloEnv env, const IloNumArray lbs, const IloNumArray ubs)`
public	`IloRangeArray(const IloEnv env, const IloIntArray lbs, const IloIntArray ubs)`

public	IloRangeArray(const IloEnv env, IloNum lb, const IloNumArray ubs)
public	IloRangeArray(const IloEnv env, const IloNumArray lbs, IloNum ub)
public	IloRangeArray(const IloEnv env, IloNum lb, const IloIntArray ubs)
public	IloRangeArray(const IloEnv env, const IloIntArray lbs, IloNum ub)

Method Summary	
public void	add(IloInt more, const IloRange range)
public void	add(const IloRange range)
public void	add(const IloRangeArray array)
public IloNumColumn	operator()(const IloNumArray vals)
public IloNumColumn	operator()(const IloIntArray vals)
public IloRange	operator[](IloInt i) const
public IloRange &	operator[](IloInt i)
public void	setBounds(const IloIntArray lbs, const IloIntArray ubs)
public void	setBounds(const IloNumArray lbs, const IloNumArray ubs)

Inherited Methods from **IloConstraintArray**
add, add, add, operator[], operator[]

Inherited Methods from **IloExtractableArray**
add, add, add, endElements, setNames

Constructors

public **IloRangeArray**(IloDefaultArrayI * i=0)

This default constructor creates an empty range array. You cannot create instances of the undocumented class IloDefaultArrayI. As an argument in this default constructor, it allows you to pass 0 (zero) as a value to an optional argument in functions and member functions that accept an array as an argument.

public **IloRangeArray**(const IloEnv env, IloInt n=0)

This constructor creates an array of n elements, each of which is an empty handle.

public **IloRangeArray**(const IloEnv env, IloInt n, IloNum lb, IloNum ub)

This constructor creates an array of n elements, each with the lower bound lb and the upper bound ub.

public **IloRangeArray**(const IloEnv env, const IloNumArray lbs, const IloNumExprArray rows, const IloNumArray ubs)

This constructor creates an array of ranges from rows, an array of expressions. It uses the corresponding elements of the arrays lbs and ubs to set the lower and upper bounds of elements in the new array. The length of rows must equal the length of lbs and ubs.

`public` **`IloRangeArray`**`(const IloEnv env, IloNum lb, const IloNumExprArray rows, const IloNumArray ubs)`

This constructor creates an array of ranges from `rows`, an array of expressions. The lower bound of every element in the new array will be `lb`. The upper bound of each element of the new array will be the corresponding element of the array `ubs`. The length of `rows` must equal the length of `ubs`.

`public` **`IloRangeArray`**`(const IloEnv env, const IloNumArray lbs, const IloNumExprArray rows, IloNum ub)`

This constructor creates an array of ranges from `rows`, an array of expressions. The upper bound of every element in the new array will be `ub`. The lower bound of each element of the new array will be the corresponding element of the array `lbs`. The length of `rows` must equal the length of `lbs`.

`public` **`IloRangeArray`**`(const IloEnv env, IloNum lb, const IloNumExprArray rows, IloNum ub)`

This constructor creates an array of ranges from `rows`, an array of expressions. The lower bound of every element in the new array will be `lb`. The upper bound of every element in the new array will be `ub`.

`public` **`IloRangeArray`**`(const IloEnv env, const IloIntArray lbs, const IloNumExprArray rows, const IloIntArray ubs)`

This constructor creates an array of ranges from `rows`, an array of expressions. It uses the corresponding elements of the arrays `lbs` and `ubs` to set the lower and upper bounds of elements in the new array. The length of `rows` must equal the length of `lbs` and `ubs`.

`public` **`IloRangeArray`**`(const IloEnv env, IloNum lb, const IloNumExprArray rows, const IloIntArray ubs)`

This constructor creates an array of ranges from `rows`, an array of expressions. The lower bound of every element in the new array will be `lb`. The upper bound of each element of the new array will be the corresponding element of the array `ubs`. The length of `rows` must equal the length of `ubs`.

`public` **`IloRangeArray`**`(const IloEnv env, const IloIntArray lbs, const IloNumExprArray rows, IloNum ub)`

This constructor creates an array of ranges from `rows`, an array of expressions. The upper bound of every element in the new array will be `ub`. The lower bound of each element of the new array will be the corresponding element of the array `lbs`. The length of `rows` must equal the length of `lbs`.

`public` **`IloRangeArray`**`(const IloEnv env, const IloNumArray lbs, const IloNumArray ubs)`

This constructor creates an array of ranges. The number of elements in the new array will be equal to the number of elements in the arrays `lbs` (or `ubs`). The number of elements in `lbs` must be equal to the number of elements in `ubs`. The lower bound of each element in the new array will be equal to the corresponding element in the array `lbs`. The upper bound of each element in the new array will be equal to the corresponding element in the array ubs.

```
public IloRangeArray(const IloEnv env, const IloIntArray lbs, const IloIntArray
ubs)
```

This constructor creates an array of ranges. The number of elements in the new array will be equal to the number of elements in the arrays `lbs` (or `ubs`). The number of elements in `lbs` must be equal to the number of elements in `ubs`. The lower bound of each element in the new array will be equal to the corresponding element in the array `lbs`. The upper bound of each element in the new array will be equal to the corresponding element in the array `ubs`.

```
public IloRangeArray(const IloEnv env, IloNum lb, const IloNumArray ubs)
```

This constructor creates an array of ranges. The number of elements in the new array will be equal to the number of elements in the array `ubs`. The lower bound of every element in the new array will be equal to `lb`. The upper bound of each element in the new array will be equal to the corresponding element in the array `ubs`.

```
public IloRangeArray(const IloEnv env, const IloNumArray lbs, IloNum ub)
```

This constructor creates an array of ranges. The number of elements in the new array will be equal to the number of elements in the array `lbs`. The upper bound of every element in the new array will be equal to `ub`. The lower bound of each element in the new array will be equal to the corresponding element in the array `lbs`.

```
public IloRangeArray(const IloEnv env, IloNum lb, const IloIntArray ubs)
```

This constructor creates an array of ranges. The number of elements in the new array will be equal to the number of elements in the array `ubs`. The lower bound of every element in the new array will be equal to `lb`. The upper bound of each element in the new array will be equal to the corresponding element in the array `ubs`.

```
public IloRangeArray(const IloEnv env, const IloIntArray lbs, IloNum ub)
```

This constructor creates an array of ranges. The number of elements in the new array will be equal to the number of elements in the array `lbs`. The upper bound of every element in the new array will be equal to `ub`. The lower bound of each element in the new array will be equal to the corresponding element in the array `lbs`.

Methods

```
public void add(IloInt more, const IloRange range)
```

This member function appends `range` to the invoking array multiple times. The argument `more` specifies how many times.

```
public void add(const IloRange range)
```

This member function appends `range` to the invoking array.

```
public void add(const IloRangeArray array)
```

This member function appends the elements in `array` to the invoking array.

`public IloNumColumn` **`operator()`**`(const IloNumArray vals)`

This operator constructs ranges in column representation. That is, it creates an instance of `IloNumColumn` that will add a newly created variable to all the ranged constraints in the invoking object, each as a linear term with the corresponding value specified in the array `values`.

`public IloNumColumn` **`operator()`**`(const IloIntArray vals)`

This operator constructs ranges in column representation. That is, it creates an instance of `IloNumColumn` that will add a newly created variable to all the ranged constraints in the invoking object, each as a linear term with the corresponding value specified in the array `values`.

`public IloRange` **`operator[]`**`(IloInt i) const`

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`. On `const` arrays, Concert Technology uses the `const` operator:

```
IloRange operator[] (IloInt i) const;
```

`public IloRange &` **`operator[]`**`(IloInt i)`

This operator returns a reference to the object located in the invoking array at the position specified by the index `i`.

`public void` **`setBounds`**`(const IloIntArray lbs, const IloIntArray ubs)`

This member function does not change the array itself; instead, it changes the bounds of all the ranged constraints that are elements of the invoking array. At the same time, it also creates an instance of the undocumented class `IloChange` to notify Concert Technology algorithms about this change in an extractable object of the model. The elements of the arrays `lbs` and `ubs` may be integer or floating-point values. The size of the invoking array must be equal to the size of `lbs` and the size of `ubs`.

Note

The member function `setBounds` notifies Concert Technology algorithms about this change of bounds for all the elements in this invoking array.

`public void` **`setBounds`**`(const IloNumArray lbs, const IloNumArray ubs)`

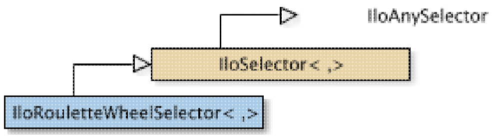
This member function does not change the array itself; instead, it changes the bounds of all the ranged constraints that are elements of the invoking array. At the same time, it also creates an instance of the undocumented class `IloChange` to notify Concert Technology algorithms about this change in an extractable object of the model. The elements of the arrays `lbs` and `ubs` may be integer or floating-point values. The size of the invoking array must be equal to the size of `lbs` and the size of `ubs`.

Note

The member function `setBounds` notifies Concert Technology algorithms about this change of bounds for all the elements in this invoking array.

Class IloRouletteWheelSelector<,>

Definition file: ilsolver/iimmulti.h
Include file: <ilsolver/iim.h>



A selector which chooses objects following a roulette wheel rule.
A *roulette wheel selection* is one where a non-negative evaluation is ascribed to each selectable object. The probability of selection of a particular object is then equal to its evaluation divided by the sums of the evaluations of the selectable objects. Roulette wheel selection chooses the *n*th object with probability `(evaluation[n] / sum(evaluation))`. This evaluation is given using an evaluator which will evaluate each object to be selected.

When you create a roulette wheel selector, you must pass a visitor object which can traverse the container from which you are selecting objects. However, in the case where a *default visitor* exists for the container in question, you can omit the visitor and the default will be used. In addition to the default visitors available in Solver, IIM provides default visitors for `IloSolutionPool` and `IloPoolProcArray`.

The following code shows how to declare a roulette wheel selector and use it to build a pool processor:

```
IloRouletteWheelSelector<IloSolution,IloSolutionPool>
  rwsel(env, parentEvaluator);
IloPoolProc selector = IloSelectSolutions(env, rwsel, IloTrue);
```

This code will select solutions based on an evaluator `parentEvaluator`, created for instance by `IloSolutionEvaluator`. The `IloTrue` parameter to `IloSelectSolutions` means that you do not want duplicates in the resulting selection. Once this selector is created, it can be used to build a reproduction goal:

```
IloGoal reproduceGoal = IloExecuteProcessor(
  env, population >> selector >> applyOp(maxOffspring) >> offspring
);
```

In this goal, the selector is fed by the population pool and will provide parent solutions to the `applyOp` processor.

Note

An instance of `IloRouletteWheelSelector` which has been transformed into a pool processor using `IloSelectSolutions` will always draw its random numbers from the random number generator of the solver on which it is executing.

See Also: IloSolutionPool, IloPoolProcArray, IloVisitor, IloEvaluator, IloExplicitEvaluator, IloMultipleEvaluator, IloUpdate, IloTournamentSelector, IloRandomSelector

Constructor Summary	
`public`	`IloRouletteWheelSelector(IloEnv env, IloEvaluator< IloObject > evaluator, IloVisitor< IloObject, IloContainer > visitor=0)` Builds a roulette wheel selector from an evaluator.

Method Summary	
`public IloEvaluator< IloObject >`	`getEvaluator() const` Delivers the evaluator given at construction time.
`public IloVisitor< IloObject, IloContainer >`	`getVisitor() const`

	Delivers the visitor used by the selector.

Inherited Methods from `IloSelector`
select

Constructors

`public IloRouletteWheelSelector(IloEnv env, IloEvaluator< IloObject > evaluator, IloVisitor< IloObject, IloContainer > visitor=0)`

Builds a roulette wheel selector from an evaluator.

This constructor builds a roulette wheel selector from an environment `env` which will evaluate its objects using an evaluator `evaluator` and an optional visitor `visitor`. Each object visited by `visitor` will be handed to `evaluator` for evaluation. If any evaluation is negative or all evaluations are zero, then an exception (of type `IloException`) is raised. If no visitor is specified, a default visitor will be used if it exists. If no default visitor exists, an exception of type `IloException` is raised.

Methods

`public IloEvaluator< IloObject > getEvaluator() const`

Delivers the evaluator given at construction time.

This member function returns the evaluator passed at construction time. If the selector was not constructed using an evaluator, then an empty handle is returned.

`public IloVisitor< IloObject, IloContainer > getVisitor() const`

Delivers the visitor used by the selector.

This member function returns the visitor passed at construction time, or, if no visitor was passed, the default visitor for the container from which object are being selected.

Class IloSearchLimit

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

An instance of this class represents a search limit in a Concert Technology model. Search limits are useful in *goals* (such as the goal returned by `IloLimitSearch` or other instances of `IloGoal`) to limit the exploration of the search tree during the search for a solution.

There are predefined functions in Concert Technology that create and return a search limit, such as `IloFailLimit`, `IloOrLimit`, and `IloTimeLimit`.

See Also: IloFailLimit, IloLimitSearch, IloOrLimit, IloTimeLimit

Constructor Summary	
public	IloSearchLimit()
public	IloSearchLimit(IloSearchLimitI * impl)

Method Summary	
public void	end() const
public IloSearchLimitI *	getImpl() const
public void	operator=(const IloSearchLimit & h)

Constructors

`public` **`IloSearchLimit`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IloSearchLimit`**`(IloSearchLimitI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

Methods

`public void` **`end`**`() const`

This member function ends the corresponding search limit and returns the memory to the environment.

`public IloSearchLimitI *` **`getImpl`**`() const`

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

`public void` **`operator=`**`(const IloSearchLimit & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IloSearchLimitI

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

The class `IloSearchLimit` represents search limits in a Concert Technology model. The class `IlcSearchLimit` represents search limits internally in a Solver search.

A search limit is used to prune part of the search tree.

A search limit is an object in Solver. Like other Solver entities, a search limit is implemented by means of two classes: a handle class and an implementation class. In other words, an instance of the class `IloSearchLimit` (a handle) contains a data member (the handle pointer) that points to an instance of the class `IloSearchLimitI` (its implementation object).

See Also: IloSearchLimit

Constructor and Destructor Summary	
public	IloSearchLimitI(IloEnvI *)
public	~IloSearchLimitI()

Method Summary	
public virtual void	display(ostream &) const
public virtual IlcSearchLimit	extract(const IloSolver solver) const
public virtual IloSearchLimitI *	makeClone(IloEnvI * env) const

Constructors and Destructors

`public` **`IloSearchLimitI`**`(IloEnvI *)`

This constructor creates an instance of the class `IloSearchLimitI`. This constructor should not be called directly as this class is an abstract class. This constructor is called automatically in the constructor of its subclasses.

`public` **`~IloSearchLimitI`**`()`

This destructor is called automatically by the destructor of its subclasses. It frees memory used by the invoking object.

Methods

`public virtual void` **`display`**`(ostream &) const`

This member function prints the invoking search limit on an output stream.

`public virtual IlcSearchLimit` **`extract`**`(const IloSolver solver) const`

In general terms, in Concert Technology, the objects of a model must be extracted for an algorithm (an instance of one of the subclasses of `IloAlgorithm`, such as `IloSolver`). This member function returns the internal search limit extracted for `solver` from the invoking search limit of a model.

`public virtual IloSearchLimitI *` **`makeClone`**`(IloEnvI * env) const`

This member function is called internally to duplicate the current search limit.

Class IloSearchSelector

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

An instance of this class is a search selector for use in a Concert Technology model. Search selectors are useful as filters during a search for a solution.

There are predefined functions in IBM® ILOG® Solver that create and return a search selector, such as `IloMinimizeVar`.

See Also: IloFirstSolution, IloMaximizeVar, IloMinimizeVar

Constructor Summary	
public	IloSearchSelector()
public	IloSearchSelector(IloSearchSelectorI * impl)

Method Summary	
public void	end() const
public IloSearchSelectorI *	getImpl() const
public void	operator=(const IloSearchSelector & h)

Constructors

`public` **`IloSearchSelector`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IloSearchSelector`**`(IloSearchSelectorI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

Methods

`public void` **`end`**`() const`

This member function ends the corresponding search selector and returns the memory to the environment.

`public IloSearchSelectorI *` **`getImpl`**`() const`

This constructor creates an object by copying another one.

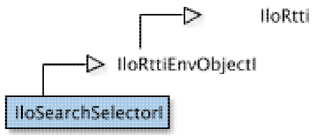
This member function returns a pointer to the implementation object of the invoking handle.

`public void` **`operator=`**`(const IloSearchSelector & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IloSearchSelectorI

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>



The class `IloSearchSelector` represents search selectors in a Concert Technology model. The class `IlcSearchSelector` represents search selectors internally in a Solver search.

Search selectors are useful to implement minimizations and filters in a Solver search.

A search selector is an object in Solver. Like other Solver entities, a search selector is implemented by means of two classes: a handle class and an implementation class. In other words, an instance of the class `IloSearchSelector` (a handle) contains a data member (the handle pointer) that points to an instance of the class `IloSearchSelectorI` (its implementation object).

See Also: IlcSearchSelector, IloSearchSelector

Constructor and Destructor Summary	
public	`IloSearchSelectorI(IloEnvI *)`
public	`~IloSearchSelectorI()`

Method Summary	
`public virtual void`	`display(ostream &) const`
`public virtual IlcSearchSelector`	`extract(const IloSolver solver) const`
`public virtual IloSearchSelectorI *`	`makeClone(IloEnvI * env) const`

Constructors and Destructors

`public` **`IloSearchSelectorI`**`(IloEnvI *)`

This constructor creates an instance of the class `IloSearchSelectorI`. This constructor should not be called directly as this class is an abstract class. This constructor is called automatically in the constructor of its subclasses.

`public` **`~IloSearchSelectorI`**`()`

This destructor is called automatically by the destructor of its subclasses. It frees memory used by the invoking object.

Methods

`public virtual void` **`display`**`(ostream &) const`

This member function prints the invoking search selector on an output stream.

```
public virtual IlcSearchSelector extract(const IloSolver solver) const
```

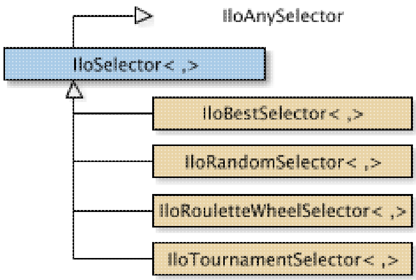
In general terms, in Concert Technology, the objects of a model must be extracted for an algorithm (an instance of one of the subclasses of `IloAlgorithm`, such as `IloSolver`). This member function returns the internal search selector extracted for `solver` from the invoking search selector of a model.

```
public virtual IloSearchSelectorI * makeClone(IloEnvI * env) const
```

This member function is called internally to duplicate the invoking search selector.

Class IloSelector<,>

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>



The IloSelector template class is the base class for all selectors. A selector IloSelector<IloObject,IloContainer> is a class that implements the selection of an object of type IloObject from a container of type IloContainer.

Selectors can easily be built with the ILOSELECTOR0 macros or by defining an instance of IloBestSelector which is a combination of a visitor, a predicate, and a comparator.

For more information, see Selectors.

See Also: ILOSELECTOR0, IloBestSelector

Method Summary	
public IloBool	select(IloObject & selected, const IloContainer & collection) const Selects an object from a collection.

Methods

public IloBool **select**(IloObject & selected, const IloContainer & collection) const

Selects an object from a collection.

This member function selects an instance of IloObject from the given IloContainer handled by the selector. If no object can be selected, IloFalse is returned and the argument selected is not assigned. If an object is selected, IloTrue is returned and selected contains the selected object.

Class IloSemaphore

Definition file: ilconcert/ilothread.h

Provides synchronization primitives.
The class `IloSemaphore` provides synchronization primitives adapted to Concert Technology. This class supports inter-thread communication in multithread applications. An instance of this class, a semaphore, is a counter; its value is always positive. This counter can be incremented or decremented. You can always increment a semaphore, and incrementing is not a blocking operation. However, the value of the counter cannot be negative, so any thread that attempts to decrement a semaphore whose counter is already equal to 0 (zero) is blocked until another thread increments the semaphore.

System Class

`IloSemaphore` is a system class.

Most Concert Technology classes are actually handle classes whose instances point to objects of a corresponding implementation class. For example, instances of the Concert Technology class `IloNumVar` are handles pointing to instances of the implementation class `IloNumVarI`. Their allocation and de-allocation in a Concert Technology environment are managed by an instance of `IloEnv`.

However, system classes, such as `IloSemaphore`, differ from that pattern. `IloSemaphore` is an ordinary C++ class. Its instances are allocated on the C++ heap.

Instances of `IloSemaphore` are not automatically de-allocated by a call to the member function `IloEnv::end`. You must explicitly destroy instances of `IloSemaphore` by means of a call to the delete operator (which calls the appropriate destructor) when your application no longer needs instances of this class.

Furthermore, you should not allocate—neither directly nor indirectly—any instance of `IloSemaphore` in a Concert Technology environment because the destructor for that instance of `IloSemaphore` will never be called automatically by `IloEnv::end` when it cleans up other Concert Technology objects in that Concert Technology environment.

For example, it is not a good idea to make an instance of `IloSemaphore` part of a conventional Concert Technology model allocated in a Concert Technology environment because that instance will not automatically be de-allocated from the Concert Technology environment along with the other Concert Technology objects.

De-allocating Instances of IloSemaphore

Instances of `IloSemaphore` differ from the usual Concert Technology objects because they are not allocated in a Concert Technology environment, and their de-allocation is not managed automatically for you by `IloEnv::end`. Instead, you must explicitly destroy instances of `IloSemaphore` by calling the delete operator when your application no longer needs those objects.

See Also: IloBarrier, IloCondition

Constructor and Destructor Summary	
`public`	`IloSemaphore(int value=0)`
`public`	`~IloSemaphore()`

Method Summary	
`public void`	`post()`
`public int`	`tryWait()`
`public void`	`wait()`

Constructors and Destructors

```
public IloSemaphore(int value=0)
```

This constructor creates an instance of `IloSemaphore`, initializes it to `value`, and allocates it on the C++ heap (not in a Concert Technology environment). If you do not pass a `value` argument, the constructor initializes the semaphore at 0 (zero).

```
public ~IloSemaphore()
```

The delete operator calls this destructor to de-allocate an instance of `IloSemaphore`. This destructor is called automatically by the runtime system. The destructor de-allocates operating system-specific data structures.

Methods

```
public void post()
```

This member function increments the invoking semaphore by 1 (one). If there are threads blocked at this semaphore, then this member function wakes one of them.

```
public int tryWait()
```

This member function attempts to decrement the invoking semaphore by 1 (one). If this decrement leaves the counter positive, then the call succeeds and returns 1 (one). If the decrement would make the counter strictly negative, then the decrement does not occur, the call fails, and the member function returns 0 (zero).

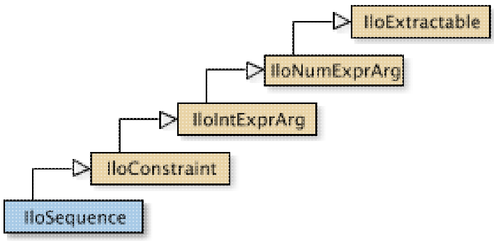
```
public void wait()
```

This member function decrements the invoking semaphore by 1 (one).

If this decrement would make the semaphore strictly negative, then this member function blocks the calling thread. The thread wakes up when the member function can safely decrement the semaphore without causing the counter to become negative (for example, if another entity increments the semaphore). If this member function cannot decrement the invoking semaphore, then it leads to deadlock.

Class IloSequence

Definition file: ilconcert/ilomodel.h



For constraint programming: a sequence constraint in a model.
An instance of this class represents a sequence constraint in a model. As you can see from the arguments of its constructor, an instance of this class makes it possible for you to constrain:

- the minimum number of allowable values in the sequence,
- the maximum number of allowable values in the sequence,
- the frequency of allowable values (that is, how often a value occurs in the sequence),
- the number of elements in the sequence.

In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloConstraint, IloDistribute

Constructor Summary	
`public`	`IloSequence()`
`public`	`IloSequence(IloSequenceI * impl)`
`public`	`IloSequence(const IloEnv env, IloInt nbMin, IloInt nbMax, IloInt seqWidth, const IloIntVarArray vars, const IloIntArray values, const IloIntVarArray cards, const char * name=0)`

Method Summary	
`public IloSequenceI *`	`getImpl() const`

Inherited Methods from `IloConstraint`
`getImpl`

Inherited Methods from `IloIntExprArg`
`getImpl`

Inherited Methods from `IloNumExprArg`
`getImpl`

Inherited Methods from `IloExtractable`
`asConstraint, asIntExpr, asModel, asNumExpr, asObjective, asVariable, end, getEnv,`

```
getId, getImpl, getName, getObject, isConstraint, isIntExpr, isModel, isNumExpr,
isObjective, isVariable, setName, setObject
```

Constructors

public **IloSequence**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IloSequence**(IloSequenceI * impl)

This constructor creates a handle object from a pointer to an implementation object.

public **IloSequence**(const IloEnv env, IloInt nbMin, IloInt nbMax, IloInt seqWidth, const IloIntVarArray vars, const IloIntArray values, const IloIntVarArray cards, const char * name=0)

This constructor creates a sequence constraint in an environment. The argument `nbMin` specifies a minimum number of allowable values, and `nbMax` specifies a maximum number of allowable values. The argument `seqWidth` specifies the number of elements in a sequence. The argument `cards` specifies an array of cardinalities (that is, how many occurrences).

In the new constraint created by this class, the constrained variables in the array `cards` will be equal to the number of occurrences in the array `vars` of the values in the array `values` such that for each sequence of `seqWidth` (a number) consecutive constrained variables of `vars`, at least `nbMin` and at most `nbMax values` are assigned to a constrained variable of the sequence.

The arrays `cards` and `values` must be the same length; otherwise, on platforms where C++ exceptions are supported and exceptions are enabled, Concert Technology throws the exception `InvalidArraysException`.

Methods

public IloSequenceI * **getImpl**() const

This member function returns a pointer to the implementation object of the invoking handle.

Class IloSolution

Definition file: ilconcert/ilosolution.h

Instances of this class store solutions to problems.
Instances of this class store solutions to problems. The fundamental property of `IloSolution` is its ability to transfer stored values from or to the active objects associated with it. In particular, the member function `IloSolution::store` stores the values from algorithm variables while the member function `IloSolution::restore` instantiates the actual variables with stored values. Variables in the solution may be selectively restored. This class also offers member functions to copy and to compare solutions.

Information about these classes of variables can be stored in an instance of `IloSolution`:

- `IloAnySet:` the required and possible sets are stored; when the variable is bound, the required and possible sets are equivalent.
- `IloAnyVar`: the value of the variable is stored.
- `IloBoolVar`: the value (true or false) of the variable is stored. Some of the member functions for `IloBoolVar` are covered by the member function for `IloNumVar`, as `IloBoolVar` is a subclass of `IloNumVar`. For example, there is no explicit member function to add objects of type `IloBoolVar`.
- `IloIntSetVar`: the required and possible sets are stored; when the variable is bound, the required and possible sets are equivalent.
- `IloNumVar`: the lower and upper bounds are stored; when the variable is bound, the current lower and upper bound are equivalent.
- `IloIntVar`: the lower and upper bounds are stored; when the variable is bound, the current lower and upper bound are equivalent.
- `IloIntervalVar`: the lower and upper bounds of the start, end, length and size are stored as well as the presence status.
- `IloObjective`: the value of the objective is stored. Objectives are never restored; operations such as `setRestorable` cannot change this. More than one instance of `IloObjective` can be added to a solution,. In such cases, there is the idea of an active objective, which is returned by `IloSolution::getObjective`. The active objective typically specifies the optimization criterion for the problem to which the solution object is a solution. For example, the IBM ILOG Solver class `IloImprove` uses the idea of an active objective.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Objects of type `IloSolution` have a scope, comprising the set of variables that have their values stored in the solution. The scope is given *before* the basic operations of storing and restoring are performed, via `add` and `remove` methods. For example,

```
IloNumVar var(env);
IloSolution soln(env);
solution.add(var);
```

creates a numeric variable and a solution and adds the variable to the solution. Arrays of variables can also be added to the solution. For example,

```
IloNumVarArray arr(env, 10, 0, 1);
soln.add(arr);
```

adds 10 variables with range [0...1]. When an array of variables is added to the solution, the array object itself is not present in the scope of the solution; only the elements are present. If the solution is then stored by means of `soln.store(algorithm)`, the values of variable `var` and `arr[0]` to `arr[9]` are saved. Any attempt to add a variable that is already present in a solution throws an exception, an instance of `IloException`.

Accessors allow access to the stored values of the variables, regardless of the state (or existence) of the algorithm they were stored from. For example,

```
cout << "arr[3] = " << soln.getValue(arr[3]) << endl;
```

Any attempt to access a variable that is not present in the solution throws an instance of `IloException`.

A variable or an array of variables can be removed from a solution. For example,

```
soln.remove(var);
```

removes `var` from the scope of the solution; and

```
soln.remove(arr);
```

removes `arr[0]` to `arr[9]` from the solution.

Any attempt to remove a variable that is not present in the solution throws an instance of `IloException`.

See Also these classes in the *IBM ILOG Solver Reference Manual*: `IloAnySetVar`, `IloAnyVar`, `IloIntSetVar`, `IloStoreSolution`, `IloFRestoreSolution`.

See Also this class in the *IBM ILOG CP Optimizer Reference Manual*: `IloIntervalVar`.

See Also: IloNumVar, IloIntVar, IloObjective

Constructor Summary	
public	IloSolution()
public	IloSolution(IloSolutionI * impl)
public	IloSolution(const IloSolution & solution)
public	IloSolution(IloEnv mem, const char * name=0)

Method Summary	
public void	add(IloAnySetVarArray a) const
public void	add(IloAnySetVar var) const
public void	add(IloAnyVarArray a) const
public void	add(IloAnyVar var) const
public void	add(IloNumVarArray a) const
public void	add(IloNumVar var) const
public void	add(IloObjective objective) const
public IloBool	contains(IloExtractable extr) const
public void	copy(IloExtractable extr, IloSolution solution) const
public void	copy(IloSolution solution) const
public void	end()
public IloEnv	getEnv() const
public IloSolutionI *	getImpl() const
public IloNum	getMax(IloNumVar var) const
public IloNum	getMin(IloNumVar var) const

public const char *	getName() const
public IloAny	getObject() const
public IloObjective	getObjective() const
public IloNum	getObjectiveValue() const
public IloNumVar	getObjectiveVar() const
public IloAnySet	getPossibleSet(IloAnySetVar var) const
public IloAnySet	getRequiredSet(IloAnySetVar var) const
public IloAny	getValue(IloAnyVar var) const
public IloNum	getValue(IloNumVar var) const
public IloNum	getValue(IloObjective obj) const
public IloBool	isBetterThan(IloSolution solution) const
public IloBool	isBound(IloAnySetVar var) const
public IloBool	isBound(IloNumVar var) const
public IloBool	isEquivalent(IloExtractable extr, IloSolution solution) const
public IloBool	isEquivalent(IloSolution solution) const
public IloBool	isObjectiveSet() const
public IloBool	isRestorable(IloExtractable extr) const
public IloBool	isWorseThan(IloSolution solution) const
public IloSolution	makeClone(IloEnv env) const
public void	operator=(const IloSolution & solution)
public void	remove(IloExtractableArray extr) const
public void	remove(IloExtractable extr) const
public void	restore(IloExtractable extr, IloAlgorithm algorithm) const
public void	restore(IloAlgorithm algorithm) const
public void	setFalse(IloBoolVar var) const
public void	setName(const char * name) const
public void	setNonRestorable(IloExtractableArray array) const
public void	setNonRestorable(IloExtractable extr) const
public void	setObject(IloAny obj) const
public void	setObjective(IloObjective objective) const
public void	setPossibleSet(IloAnySetVar var, IloAnySet possible) const
public void	setRequiredSet(IloAnySetVar var, IloAnySet required) const
public void	setRestorable(IloExtractableArray array) const
public void	setRestorable(IloExtractable ex) const
public void	setTrue(IloBoolVar var) const
public void	setValue(IloAnyVar var, IloAny value) const
public void	setValue(IloObjective objective, IloNum value) const
public void	store(IloExtractable extr, IloAlgorithm algorithm) const
public void	store(IloAlgorithm algorithm) const
public void	unsetObjective() const

Inner Class	
IloSolution::Iterator	It allows you to traverse the variables in a solution.

Constructors

`public IloSolution()`

This constructor creates a solution whose implementation pointer is 0 (zero). The handle must be assigned before its methods can be used.

`public IloSolution(IloSolutionI * impl)`

This constructor creates a handle object (an instance of the class `IloSolution`) from a pointer to an implementation object (an instance of the class `IloSolutionI`).

`public IloSolution(const IloSolution & solution)`

This constructor creates a handle object from a reference to a solution. After execution, both the newly constructed handle and `solution` point to the same implementation object.

`public IloSolution(IloEnv mem, const char * name=0)`

This constructor creates an instance of the `IloSolution` class. The optional argument `name`, if supplied, becomes the name of the created object.

Methods

`public void add(IloAnySetVarArray a) const`

This member function adds each element of `array` to the invoking solution.

`public void add(IloAnySetVar var) const`

This member function adds the set variable `var` to the invoking solution.

`public void add(IloAnyVarArray a) const`

This member function adds each element of `array` to the invoking solution.

`public void add(IloAnyVar var) const`

This member function adds the variable `var` to the invoking solution.

`public void add(IloNumVarArray a) const`

This member function adds each element of `array` to the invoking solution.

`public void **add**(IloNumVar var) const`

This member function adds the variable `var` to the invoking solution.

`public void **add**(IloObjective objective) const`

This member function adds `objective` to the invoking solution. If the solution has no active objective, then `objective` becomes the active objective. Otherwise, the active objective remains unchanged.

`public IloBool **contains**(IloExtractable extr) const`

This member function returns `IloTrue` if `extr` is present in the invoking object. Otherwise, it returns `IloFalse`.

`public void **copy**(IloExtractable extr, IloSolution solution) const`

This member function copies the saved value of `extr` from `solution` to the invoking solution. If `extr` does not exist in either `solution` or the invoking object, this member function throws an instance of `IloException`. The restorable status of `extr` is not copied.

`public void **copy**(IloSolution solution) const`

For each variable that has been added to `solution`, this member function copies its saved data to the invoking solution. If a particular extractable does not already exist in the invoking solution, it is automatically added first. If variables were added to the invoking solution, their restorable status is the same as in `solution`. Otherwise, their status remains unchanged in the invoking solution.

`public void **end**()`

This member function deallocates the memory used to store the solution. If you no longer need a solution, calling this member function can reduce memory consumption.

`public IloEnv **getEnv**() const`

This member function returns the environment specified when the invoking object was constructed.

`public IloSolutionI * **getImpl**() const`

This member function returns a pointer to the implementation object corresponding to the invoking solution.

`public IloNum **getMax**(IloNumVar var) const`

This member function returns the maximal value of the variable `var` in the invoking solution.

`public IloNum` **`getMin`**`(IloNumVar var) const`

This member function returns the minimal value of the variable `var` in the invoking solution.

`public const char *` **`getName`**`() const`

This member function returns a character string specifying the name of the invoking object (if there is one).

`public IloAny` **`getObject`**`() const`

This member function returns the object associated with the invoking object (if there is one). Normally, an associated object contains user data pertinent to the invoking object.

`public IloObjective` **`getObjective`**`() const`

This member function returns the *active* objective as set via a previous call to `IloSolution::add` or `setObjective(IloObjective)`. If there is no active objective, an empty handle is returned.

`public IloNum` **`getObjectiveValue`**`() const`

This member function returns the saved value of the current active objective. It can be seen as performing the action `getValue(getObjective())`.

`public IloNumVar` **`getObjectiveVar`**`() const`

If the active objective corresponds to a simple `IloNumVar`, this member function returns that variable. If there is no active objective or if the objective is not a simple variable, an empty handle is returned.

`public IloAnySet` **`getPossibleSet`**`(IloAnySetVar var) const`

This member function returns the set of possible values for the variable `var`, as stored in the invoking solution.

`public IloAnySet` **`getRequiredSet`**`(IloAnySetVar var) const`

This member function returns the set of required values for the variable `var`, as stored in the invoking solution.

`public IloAny` **`getValue`**`(IloAnyVar var) const`

This member function returns the value of the variable `var` in the invoking solution.

`public IloNum` **`getValue`**`(IloNumVar var) const`

This member function returns the value of the variable `var` in the invoking solution. If the saved minimum and maximum of the variable are not equal, this member function throws an instance of `IloException`.

`public IloNum **getValue**(IloObjective obj) const`

This member function returns the saved value of objective `objective` in the invoking solution.

`public IloBool **isBetterThan**(IloSolution solution) const`

This member function returns `IloTrue` if the invoking solution and `solution` have the same objective and if the invoking solution has a strictly higher quality objective value (according to the sense of the objective). In all other situations, it returns `IloFalse`.

`public IloBool **isBound**(IloAnySetVar var) const`

This member function returns `IloTrue` if the stored required and possible sets for the set variable `var` are equal in the invoking solution. Otherwise, it returns `IloFalse`.

`public IloBool **isBound**(IloNumVar var) const`

This member function returns `IloTrue` if `var` takes a single value in the invoking solution. Otherwise, it returns `IloFalse`.

`public IloBool **isEquivalent**(IloExtractable extr, IloSolution solution) const`

This member function returns `IloTrue` if the saved value of `extr` is the same in the invoking solution and `solution`. Otherwise, it returns `IloFalse`. If `extr` does not exist in either `solution` or the invoking object, the member function throws an instance of `IloException`.

`public IloBool **isEquivalent**(IloSolution solution) const`

This member function returns `IloTrue` if the invoking object and `solution` contain the same variables with the same saved values. Otherwise, it returns `IloFalse`.

`public IloBool **isObjectiveSet**() const`

This member function returns `IloTrue` if the invoking solution has an active objective. Otherwise, it returns `IloFalse`.

`public IloBool **isRestorable**(IloExtractable extr) const`

This member function returns `IloFalse` if `setNonRestorable(extr)` was called more recently than `setRestorable(extr)`. Otherwise, it returns `IloTrue`. This member function always returns `IloFalse` when it is passed an `IloObjective` object.

`public IloBool isWorseThan(IloSolution solution) const`

This member function returns `IloTrue` if the invoking solution and `solution` have the same objective and if the invoking solution has a strictly lower quality objective value (according to the sense of the objective). In all other situations, it returns `IloFalse`.

`public IloSolution makeClone(IloEnv env) const`

This member function allocates a new solution on `env` and adds to it all variables that were added to the invoking object. The "restorable" status of all variables in the clone is the same as that in the invoking solution. Likewise, the active objective in the clone is the same as that in the invoking solution. The newly created solution is returned.

`public void operator=(const IloSolution & solution)`

This operator assigns an address to the handle pointer of the invoking solution. That address is the location of the implementation object of `solution`. After the execution of this operator, the invoking solution and `solution` both point to the same implementation object.

`public void remove(IloExtractableArray extr) const`

This member function removes each element of `array` from the invoking solution. If the invoking solution does not contain all elements of `array`, the member function throws an instance of `IloException`.

`public void remove(IloExtractable extr) const`

This member function removes extractable `extr` from the invoking solution. If the invoking solution does not contain `extr`, the member function throws an instance of `IloException`.

`public void restore(IloExtractable extr, IloAlgorithm algorithm) const`

This member function restores the value of the extractable corresponding to `extr` by reference to `algorithm`. The use of this member function depends on the state of `algorithm`. If `algorithm` is an instance of the IBM ILOG Solver class `IloSolver`, this member function can only be used during search. If `extr` does not exist in the invoking solution, the member function throws an instance of `IloException`.

`public void restore(IloAlgorithm algorithm) const`

This member function uses `algorithm` to instantiate the variables in the invoking solution with their saved values. The value of any objective added to the solution is not restored. The use of this member function depends on the state of `algorithm`. If `algorithm` is an instance of the IBM ILOG Solver class `IloSolver`, this member function can only be used during search.

`public void setFalse(IloBoolVar var) const`

This member function sets the stored value of `var` to `IloFalse` in the invoking solution.

`public void setName(const char * name) const`

This member function assigns `name` to the invoking object.

`public void setNonRestorable(IloExtractableArray array) const`

This member function specifies to the invoking solution that when the solution is restored by means of `restore(IloAlgorithm)` or `restore(IloExtractable, IloAlgorithm)`, no elements of `array` will be restored. When an array of variables is added to a solution, each variable is added in a “restorable” state.

`public void setNonRestorable(IloExtractable extr) const`

This member function specifies to the invoking solution that when the solution is restored by means of `restore(IloAlgorithm)` or `restore(IloExtractable, IloAlgorithm)`, `extr` will not be restored. When a variable is added to a solution, it is added in a “restorable” state.

`public void setObject(IloAny obj) const`

This member function associates `obj` with the invoking object. The member function `getObject` accesses this associated object afterward. Normally, `obj` contains user data pertinent to the invoking object.

`public void setObjective(IloObjective objective) const`

This member function adds `objective` to the invoking solution, if it is not already present, and sets the active objective to `objective`.

`public void setPossibleSet(IloAnySetVar var, IloAnySet possible) const`

This member function sets the stored possible values for `var` as `possible` in the invoking solution.

`public void setRequiredSet(IloAnySetVar var, IloAnySet required) const`

This member function sets the stored required values for `var` as `required` in the invoking solution.

`public void setRestorable(IloExtractableArray array) const`

This member function specifies to the invoking solution that when the solution is restored by means of `restore(IloAlgorithm)` or `restore(IloExtractable, IloAlgorithm)`, the appropriate element(s) of `array` will be restored. When an array of variables is added to a solution, each variable is added in a “restorable” state. This call has no effect on objects of type `IloObjective`; objects of this type are never restored.

`public void setRestorable(IloExtractable ex) const`

This member function specifies to the invoking solution that when the solution is restored by means of `restore(IloAlgorithm)` or `restore(IloExtractable, IloAlgorithm)`, `extr` will be restored. When a variable is added to a solution, it is added in a “restorable” state. This call has no effect on objects of type `IloObjective`; objects of that type are never restored.

`public void` **`setTrue`**`(IloBoolVar var) const`

This member function sets the stored value of `var` to `IloTrue` in the invoking solution.

`public void` **`setValue`**`(IloAnyVar var, IloAny value) const`

This member function sets the value of the variable `var` to `value` in the invoking solution.

`public void` **`setValue`**`(IloObjective objective, IloNum value) const`

This member function sets the value of `objective` as stored in the invoking solution to `value`. This member function should be used with care and only when the objective value of the solution is known exactly.

`public void` **`store`**`(IloExtractable extr, IloAlgorithm algorithm) const`

This member function stores the value of the extractable corresponding to `extr` by reference to `algorithm`. If `extr` does not exist in the invoking solution, the member function throws an instance of `IloException`.

`public void` **`store`**`(IloAlgorithm algorithm) const`

This member function stores the values of the objects added to the solution by reference to `algorithm`.

`public void` **`unsetObjective`**`() const`

This member function asserts that there should be no active objective in the invoking solution, although the previous active object is still present. A new active objective can be set via `IloSolution::add` or `IloSolution::setObjective`.

Class IloSolutionDeltaCheck

Definition file: ilsolver/iimmeta.h
Include file: <ilsolver/iimls.h>

In some instances, the illegality of a solution delta can be determined before the delta is applied to the solution in question, if knowledge of the problem is available. This handle class can be used to perform such “pre-filtering”, which can considerably improve the performance of local search procedures. `IloMetaHeuristic::getDeltaCheck` returns such a delta checker. This is the most common use of the class.

See Also: IloMetaHeuristic, IloScanNHood, IloSolutionDeltaCheckI

Constructor Summary	
public	IloSolutionDeltaCheck()
public	IloSolutionDeltaCheck(IloSolutionDeltaCheckI * ck)

Method Summary	
public IloEnv	getEnv() const
public IloSolutionDeltaCheckI *	getImpl() const
public IloBool	ok(IloSolver solver, IloSolution delta)
public void	operator=(const IloSolutionDeltaCheck & check)

Constructors

public **IloSolutionDeltaCheck**()

This constructor creates a delta checker whose implementation pointer is 0 (zero). The handle must be assigned before its methods can be used.

public **IloSolutionDeltaCheck**(IloSolutionDeltaCheckI * ck)

This constructor creates a handle object (an instance of the class `IloSolutionDeltaCheck` from a pointer to an implementation object (an instance of the class `IloSolutionDeltaCheckI`).

Methods

public IloEnv **getEnv**() const

This member function returns the environment specified when the implementation object of the invoking object was constructed.

public IloSolutionDeltaCheckI * **getImpl**() const

This member function returns a pointer to the implementation object corresponding to the invoking delta checker.

`public IloBool` **`ok`**`(IloSolver solver, IloSolution delta)`

This member function is called with the delta to be checked. If `delta` is determined to be illegal, this member function returns `IloFalse`. Otherwise, it returns `IloTrue`.

`public void` **`operator=`**`(const IloSolutionDeltaCheck & check)`

This operator assigns an address to the handle pointer of the invoking delta checker. That address is the location of the implementation object of `check`. After the execution of this operator, the invoking delta checker and `check` both point to the same implementation object.

Class IloSolutionDeltaCheckI

Definition file: ilsolver/iimmeta.h
Include file: <ilsolver/iimls.h>

In some instances, the illegality of a solution delta can be determined before the delta is applied to the solution in question, if knowledge of the problem is available. This abstract implementation class can be used to perform such specialized “pre-filtering”, which can improve the performance of local search procedures considerably.

The method `IloMetaHeuristic::getDeltaCheck` returns a delta checker which uses the method `IloMetaHeuristicI::isFeasible` to perform such checks. Therefore, it is not necessary to subclass `IloSolutionDeltaCheck` to perform delta checks for metaheuristics.

See Also: IloMetaHeuristic, IloScanNHood, IloSolutionDeltaCheck

Constructor and Destructor Summary	
public	IloSolutionDeltaCheckI(IloEnv env)
public	~IloSolutionDeltaCheckI()

Method Summary	
public IloEnv	getEnv() const
public virtual IloBool	ok(IloSolver solver, IloSolution delta) const

Constructors and Destructors

`public` **`IloSolutionDeltaCheckI`**`(IloEnv env)`

This constructor creates an instance of the `IloSolutionDeltaCheckI` class.

`public` **`~IloSolutionDeltaCheckI`**`()`

Since `IloSolutionDeltaCheckI` is an abstract class, a virtual destructor is provided.

Methods

`public IloEnv` **`getEnv`**`() const`

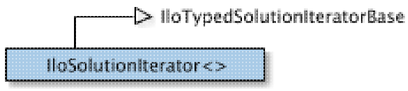
This member function returns the environment specified when the invoking object was constructed.

`public virtual IloBool` **`ok`**`(IloSolver solver, IloSolution delta) const`

This member function is called with the delta to be checked. If `delta` is determined to be illegal, this member should return `IloFalse`. Otherwise, it returns `IloTrue`.

Class IloSolutionIterator<>

Definition file: ilconcert/ilosolution.h



This template class creates a typed iterator over solutions.
You can use this iterator to discover all extractable objects added to a solution and of a particular type. The type is denoted by `E` in the template.

This iterator is not robust. If the variable at the current position is deleted from the solution being iterated over, the behavior of this iterator afterward is undefined.

An iterator created with this template differs from an instance of `IloSolution::Iterator`. An instance of `IloSolution::Iterator` works on all extractable objects within a given solution (an instance of `IloSolution`). In contrast, an iterator created with this template only iterates over extractable objects of the specified type.

See Also: IloSolution, IloSolution::Iterator

Constructor Summary	
public	IloSolutionIterator(IloSolution s)

Method Summary	
public E	operator*() const
public void	operator++()

Constructors

`public` **`IloSolutionIterator`**`(IloSolution s)`

This constructor creates an iterator for instances of the class `E`.

Methods

`public E` **`operator*`**`() const`

This operator returns the current element, the one to which the invoking iterator points. This current element is a handle to an extractable object (not a pointer to the implementation object).

`public void` **`operator++`**`()`

This operator advances the iterator by one position.

Class IloSolutionManip

Definition file: ilconcert/ilosolution.h

An instance of this class accesses a specific part of a solution.
To display a specific part of the solution, you construct the class `IloSolutionManip` from a solution and an extractable object. You use the `operator<<` with this constructed class to display information stored on the specified extractable object in the solution.

See Also: IloSolution, operator<<

Constructor Summary	
`public`	`IloSolutionManip(IloSolution solution, IloExtractable extr)`

Constructors

`public` **`IloSolutionManip`**`(IloSolution solution, IloExtractable extr)`

This constructor creates an instance of `IloSolutionManip` from the solution specified by `solution` and from the extractable object `extr`. The constructor throws an exception (an instance of `IloException`) if `extr` has not been added to `solution`. You can use the `operator<<` with the newly created object to display the information in `extr` stored in `solution`.

Class IloSolutionPool

Definition file: ilsolver/iimpool.h
Include file: <ilsolver/iim.h>

A pool of solutions.
Solution pools are bags of solutions which can be used:

- as containers (for example for holding a population of solutions)
- as communication buffers between pool processors

For example, the following code uses the `buffer` pool to store intermediate solutions which will be processed by the `consumer` processor, then stored in the `result` pool:

```
IloSolutionPool buffer(env);
IloSolutionPool result(env);
IloPoolProc producer = ...; // a processor which generates solutions
IloPoolProc consumer = ...; // a processor which transforms solutions

IloPoolProc proc = producer >> buffer >> consumer >> result;
```

Note

IIM provides a default visitor for `IloSolutionPool`. Thus, selection of a solution from a pool can be performed without specification of a visitor.

See Also: IloPoolProc, operator>>, IloVisitor, IloSelector, IloSelectSolutions, IloReplaceSolutions, IloSolution

Constructor Summary	
public	IloSolutionPool(const IloAnyPool & obj)
public	IloSolutionPool(IloEnv env, const char * name=0)

Method Summary	
public void	add(IloSolution elt) const
public void	addAll(IloSolutionPool pool) const
public void	addListener(IloListener listener) const
public void	copy(IloSolutionPool pool) const
public void	end()
public void	endSolutions() const
public IloComparator< IloSolution >	getDefaultComparator() const
public IloEnv	getEnv() const
public IloInt	getSize() const
public IloComparator< IloSolution >	getSortComparator() const
public void	remove(IloInt index) const
public void	remove(IloSolution elt) const
public void	removeAll() const

public void	removeAll(IloSolutionPool pool) const
public void	removeListener(IloListener listener) const
public void	setSortComparator(IloComparator< IloSolution > cmp) const
public void	sort() const
public void	sort(IloComparator< IloSolution > cmp) const

Inner Class
IloSolutionPool::AddedEvent
IloSolutionPool::EndEvent
IloSolutionPool::Event
IloSolutionPool::Iterator
IloSolutionPool::RemovedEvent

Constructors

public **IloSolutionPool**(const IloAnyPool & obj)

brief Constructs an `0` from an `1`.

Type checking is performed to make sure that `obj` was constructed as an `0`. If this was not the case an exception is thrown (an instance of `IloException`).

public **IloSolutionPool**(IloEnv env, const char * name=0)

brief Creates a pool for storing objects of type `1`.

This constructor creates a pool for storing objects of type `1` on the environment `env`.

Methods

public void **add**(IloSolution elt) const

brief Adds an element to the pool.

This function adds an element `elt`, an instance of `1`, to the pool.

public void **addAll**(IloSolutionPool pool) const

brief Adds all the instances of `1` contained in the given pool to the invoking pool.

This function adds all the instances of `1` in the pool `pool` to the invoking pool.

public void **addListener**(IloListener listener) const

brief Adds a listener to the pool.

This function adds a listener `listener` to the pool. According to the event type of the listener, the listener will be called when the pool has an element added or deleted, or when the pool itself is destroyed.

See Also: ILOIIMLISTENER0

```
public void copy(IloSolutionPool pool) const
```

brief Copies elements from another pool into the invoking pool.

This function copies elements from the pool `pool` into the invoking pool. All elements previously in the invoking pool will first be removed. After execution, both pools will contain the same elements.

```
public void end()
```

brief Destroy the pool.

This function destroys the invoking pool, but not its contents.

```
public void endSolutions() const
```

brief Destroys all items contained in this `0`.

This function destroys all items contained in this `0`.

```
public IloComparator< IloSolution > getDefaultComparator() const
```

brief Returns the default object comparator which is inherent to the type of pool.

This function returns the default object comparator which is inherent to the type of pool. Normally, you would never call this function directly. It is typically called from `getSortComparator` when no sorting comparator has been set on the pool. For example, `IloSolutionPool` delivers `IloBestSolutionComparator` here.

See Also: IloSolutionPool::setSortComparator, IloSolutionPool::getSortComparator, IloBestSolutionComparator

```
public IloEnv getEnv() const
```

brief Delivers the environment passed at construction time.

This function returns the environment passed in the constructor.

```
public IloInt getSize() const
```

brief Delivers the number of elements comprising the pool.

This function returns the number of elements in the pool.

```
public IloComparator< IloSolution > getSortComparator() const
```

brief Returns the comparator used for comparing elements of the pool.

This function returns the comparator previously set using `setSortComparator`. If no sorting comparator has been set, then the comparator returned is that from `getDefaultComparator`.

`public void remove(IloInt index) const`

brief Removes the `l` located at the given index from the pool.

This function removes the instance of `l` located at the given index `index` from the pool.

`public void remove(IloSolution elt) const`

brief Removes the given `l` from the pool.

This function removes `elt`, an instance of `l`, from the pool.

`public void removeAll() const`

brief Removes all the instances of `l` from the pool.

This function removes all the instances of `l` from the pool.

`public void removeAll(IloSolutionPool pool) const`

brief Removes from this pool all the instances of `l` contained in the given pool.

This function removes all the instances of `l` contained in the pool `pool`.

`public void removeListener(IloListener listener) const`

brief Removes a listener from a pool.

This function removes the listener `listener` from the invoking pool. After execution of this member function, `listener` will no longer be called when the pool is modified or destroyed.

`public void setSortComparator(IloComparator< IloSolution > cmp) const`

brief Sets the comparator used for comparing elements of the pool.

This function sets the comparator `cmp` used for comparing elements of the pool. The comparator set by this method will be used by the pool for comparing elements, particularly during the sorting of the pool, and in retrieving the best and worst elements.

See Also: IloSolutionPool::getSortComparator, IloSolutionPool::sort, IloComparator

`public void sort() const`

brief Sorts the pool using the comparator returned from `getSortComparator`.

This function sorts the pool using the comparator returned from `getSortComparator`. After sorting, the indices of the sorted elements are smaller for preferred elements. In other words, the objects are ranked "best first".

See Also: IloSolutionPool::getSortComparator

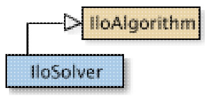
```
public void sort(IloComparator< IloSolution > cmp) const
```

brief Sorts the pool using the given comparator.

This function sorts the pool using the comparator `cmp`. After sorting, the indices of the sorted elements are smaller for preferred elements. In other words, the objects are ranked "best first".

Class IloSolver

Definition file: ilsolver/ilosolverhandle.h
Include file: <ilsolver/ilosolver.h>



An instance of this class represents an algorithm for IBM® ILOG® Solver.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

Choice Functions and Criteria

An instance of `IloSolver` allows you to set parameters to control the order in which constrained variables are bound to values. You control these search primitives by means of choice functions and criteria. In Concert Technology, the classes listed in `IloChoose` support these search primitives in a model.

Enumeration Algorithms

There are enumeration algorithms available in an instance of `IloSolver`. With IBM ILOG Concert Technology, you access those enumeration algorithms most easily through predefined goals, such as `IloBestGenerate`, `IloBestInstantiate`, `IloGenerate`, and `IloInstantiate`, in a model.

Modifying the Search Globally

These member functions have an overall effect on the search performed by `IloSolver::solve`. They allow you to limit the number of failures, the number of choice points, or the amount of time spent in a search.

```
void setFailLimit (IloInt numberOfFailures) const;
void setTimeLimit (IloNum time) const;
void setOrLimit (IloInt numberOfChoicePoints) const;
```

The following member function lets you control the granularity of the optimization:

```
void setOptimizationStep (IloNum step) const;
```

Nonlinear Propagation Techniques

When you use the member function `IloSolver::useNonLinConstraint`, the invoking solver will use nonlinear propagation techniques for all the nonlinear constraints in your model.

In addition, when the set of nonlinear constraints is a square system of equations or a square system of inequalities, a global constraint will be posted. This constraint may allow for additional propagation.

Furthermore it may allow you to *prove* that the solution-interval found for the system of nonlinear constraints surely contains a solution.

The `keep` parameter of `IloSolver::useNonLinConstraint`specifies whether or not in addition to the advanced propagation techniques the invoking solver should also use its default propagation. If `keep` is `IlcTrue` constraints are propagated only by the nonlinear propagation techniques; if `keep` is `IlcFalse`, the invoking solver will use nonlinear propagation techniques as well as its default propagation techniques.

By default, when you use nonlinear propagation techniques, the default precision (indicated by `IloSolver::getDefaultPrecision` and changed by `IloSolver::setDefaultPrecision`) is respected. However, you can set your own value to use with nonlinear constraints and to use in the safety of solutions by means of the member function `IloSolver::setNonLinPrecision` and

IloSolver::getNonLinPrecision.

Example

Here is an example using nonlinear propagation techniques.

```
#include <ilconcert/ilomodel.h>
#include <ilconcert/ilosolver.h>

 ILOSTLBEGIN

 int main () {
   try {
     IloEnv env;
     IloModel model(env);

     IloNumVar x(env,-10, 10);
     IloNumVar y(env,-10, 10);
     IloNumVarArray vars(env, 2, x, y);

     model.add(IloSquare(x) + IloSquare(y) == 1);
     model.add(IloSquare(x) == y);

     IloSolver solver(env);
     solver.useNonLinConstraint();
     solver.extract(model);

     solver.startNewSearch(IloSplit(env, vars));

     IloInt nbSol = 0;
     while (solver.next()) {
       nbSol++;
       solver.out() << ?Sol#? << nbSol << ?  ?;
       if (solver.isNonLinSafe())
  solver.out() << ?[SAFE] ?;
       solver.out() << endl;
       solver.out() << ?x: ? << solver.getFloatVar(x) <<
 endl;
       solver.out() << ?y: ? << solver.getFloatVar(y) <<
 endl;
       solver.out() << endl;
     }
     env.end();
   }
   catch (IloException& ex) {
     cerr << ?Error: ? << ex << endl;
   }
   return 0;
 }
```

Here is the result of that program:

```
 Sol#1       [SAFE]
 x: [0.786151..0.786152]
 y: [0.618033..0.618034]

 Sol#2       [SAFE]
 x: [-0.786152..-0.786151]
 y: [0.618033..0.618034]
```

Safe Solutions

When it is working with constrained floating-point variables, an instance of IloSolver returns an interval or set of intervals possibly containing a solution. Practical considerations about the representation of real numbers as floating-point values on any computing platform make it difficult to predict whether an interval of a given precision contains a unique solution (that is, an assignment of values to variables that satisfies the constraints of the problem).

However, in some cases, the algorithms of `IloSolver` can prove the existence of solutions within precisely defined intervals. In its search for solutions, an instance of `IloSolver` regards a solution as *safe* if it is certain that a solution exists within the most recently returned intervals. When it proves a solution safe, it considers the model most recently synchronized within a call to the member functions `IloSolver::solve` or `IloSolver::startNewSearch`.

The member function `IloSolver::isNonLinSafe` applies to the non linear constraints of a problem; it tells you whether Solver has proved the existence of a solution with respect to the non linear constraints of a problem within the most recently returned set of intervals. The return value of this member function applies only to the non linear constraints of a problem; that is, it can prove that the set of non linear constraints has a solution, but it says nothing about any other constraints in the problem. In some cases, there may be other constraints that are *not* non linear (constraints such as `IloAllDiff`, for example; in other words, constraints that are not non linear equalities or inequalities) in the problem that make it impossible to find a solution to the problem in its entirety.

Exceptions

Early versions of Solver raised errors in case of anomalous situations. Those errors were associated with an error code enumerated in `IlcErrorType`. As of version 5.0, an instance of `IloSolver` now silently transforms an error into an exception, that is, an instance of `IloSolver::SolverErrorException`. The member function `IloSolver::SolverErrorException::getErrorType` will return the `IlcErrorType`.

See Also: IloAddConstraint, IloSynchronizeMode, IloSolver::FailureStatus

Constructor Summary	
public	IloSolver(const IloEnv & env)
public	IloSolver(const IloModel & model)

Method Summary	
public void	add(const IlcConstraintArray constraints) const
public void	add(const IlcConstraint constraint) const
public void	addMemoryManager(IlcMemoryManagerI * mm) const
public void	addReversibleAction(const IlcGoal goal) const
public void	addTrace(IloCPTrace search)
public IloBool	assumeStrictNumericalDivision() const
public void	assumeStrictNumericalDivision(IloBool snd) const
public void	commitSearch(IloAny label) const
public void	end()
public void	endSearch() const
public void	exitSearch() const
public void	extract(const IloModel model) const
public void	fail(IlcAny label=0) const
public IlcDemon	getActiveDemon() const
public IlcGoal	getActiveGoal() const
public IlcAnyArray	getAnyArray(const IloAnyArray arg) const
public IlcAnySet	getAnySet(const IloAnySet arg) const
public IloAnySet	getAnySetValue(const IloAnySetVar var) const
public IlcAnySetVar	getAnySetVar(const IloAnySetVar var) const
public IlcAnySetVarArray	getAnySetVarArray(const IloAnySetVarArray vars) const

public IloAny	getAnyValue(const IloAnyVar var) const
public IlcAnyVar	getAnyVar(const IloAnyVar var) const
public IlcAnyVarArray	getAnyVarArray(const IloAnyVarArray vars) const
public IlcBox	getBox(const IloBox) const
public IloNum	getBranchImpact(const IloNumVar x) const
public IloNum	getBranchImpact(const IlcFloatVar x) const
public IloNum	getBranchImpact(const IloIntVar x) const
public IloNum	getBranchImpact(const IlcIntVar x) const
public IlcConstraint	getConstraint(const IloConstraint ct) const
public IlcInt	getCurrentNumberOfNodes() const
public IlcFloat	getDefaultPrecision() const
public IloInt	getDegree(const IloIntVar x) const
public IloInt	getDegree(const IlcIntVar x) const
public IlcFloat	getElapsedTime() const
public IloSolver::FailureStatus	getFailureStatus() const
public IlcFloatArray	getFloatArray(const IloNumArray arg) const
public IlcFloatDisplay	getFloatDisplay() const
public IlcFloatExp	getFloatExp(const IloNumExprArg expr) const
public IlcFloatVar	getFloatVar(const IloNumVar var) const
public IlcFloatVarArray	getFloatVarArray(const IloNumVarArray vars) const
public IlcAllocationStack *	getHeap() const
public IlcFloat	getImpact(const IlcIntVar x) const
public IlcFloat	getImpact(const IlcIntVar x, IlcInt v, IlcBool countFails=IlcTrue) const
public IloSolverI *	getImpl() const
public IlcIntArray	getIntArray(const IloNumArray arg) const
public IlcIntExp	getIntExp(const IloIntExprArg expr) const
public IlcIntSet	getIntSet(const IloIntSet arg) const
public IloIntSet	getIntSetValue(const IloIntSetVar var) const
public IlcIntSetVar	getIntSetVar(const IloIntSetVar var) const
public IlcIntSetVarArray	getIntSetVarArray(const IloNumSetVarArray vars) const
public IlcIntVar	getIntVar(const IloNumVar var) const
public IlcIntVarArray	getIntVarArray(const IloNumVarArray vars) const
public IlcFloat	getLocalImpact(const IlcIntVar x, IlcInt v) const
public IlcFloat	getLocalVarImpact(const IlcIntVar x, IlcInt depth=-1) const
public IloNum	getMax(const IloNumVar v) const
public IlcUInt	getMaxSearchMemoryUsage() const
public IlcUInt	getMemoryUsage() const
public IloNum	getMin(const IloNumVar v) const
public IlcFloat	getNonLinPrecision() const

public IlcInt	getNumberOfChoicePoints() const
public IlcInt	getNumberOfConstraints() const
public IlcInt	getNumberOfCuts() const
public IlcFloat	getNumberOfFails(const IlcIntVar x, IlcInt v) const
public IlcInt	getNumberOfFails() const
public IlcInt	getNumberOfFiles() const
public IlcFloat	getNumberOfInstantiations(const IlcIntVar x, IlcInt v) const
public IlcInt	getNumberOfMoves() const
public IlcInt	getNumberOfRecomputed() const
public IlcInt	getNumberOfVariables() const
public IloNum	getOptimizationStep() const
public IlcAllocationStack *	getPersistentHeap() const
public IlcRandom	getRandom() const
public IloNum	getReduction(const IloNumVar x) const
public IloNum	getReduction(const IlcFloatVar x) const
public IloInt	getReduction(const IloIntVar x) const
public IloInt	getReduction(const IlcIntVar x) const
public IloNum	getRelativeOptimizationStep() const
public IlcUInt	getSearchMemoryUsage() const
public IlcSearchNode	getSearchNode() const
public IlcFloat	getSuccessRate(const IlcIntVar x) const
public IlcFloat	getSuccessRate(const IlcIntVar x, IlcInt v) const
public IlcFloat	getTime() const
public IlcInt	getTotalNumberOfNodes() const
public IloNum	getValue(const IloNumVar v) const
public IlcInt	getWorkerId() const
public IloBool	isExtracted(const IloExtractable ext) const
public IloBool	isInRecomputeMode() const
public IloBool	isInSearch() const
public IloBool	isInteger(const IloNumVar var) const
public IloBool	isNonLinSafe() const
public IloBool	next() const
public void	printInformation(ostream & stream) const
public void	printInformation() const
public IloBool	propagate(const IloConstraint constraint=0, IloSynchronizeMode mode=IloSynchronizeAndRestart, IloBool restore=IloFalse) const
public void	removeTrace(IloCPTrace search)
public void	restartSearch() const
public void	setBigTupleSet() const
public void	

	setDefaultFilterLevel(IlcFilterLevelConstraint ct, IlcFilterLevel level) const
public void	setDefaultPrecision(IloNum precision) const
public void	setFailLimit(IloInt numberOfFailures) const
public void	setFastRestartMode(IloBool mode) const
public void	setFileNodeOptions(IlcInt maxSize, char * prefixName, IlcBool useCompression, IlcBool useDisk) const
public void	setFilterLevel(IlcConstraint ct, IlcFilterLevel level) const
public void	setFilterLevel(IloConstraint oct, IlcFilterLevel level) const
public void	setFloatDisplay(IlcFloatDisplay display) const
public void	setHoloTupleSet() const
public void	setMax(const IloNumVar v, IloNum n) const
public void	setMin(const IloNumVar v, IloNum n) const
public void	setMonitor(const IlcSearchMonitor monitor, IloBool rec) const
public void	setNonLinPrecision(IlcFloat p) const
public void	setObjMin(const IlcFloatVar & obj, IloNum step, IloNum r=0.0) const
public void	setObjMin(const IlcIntVar & obj, IloInt step=1, IloNum r=0.0) const
public void	setOptimizationStep(IloNum step) const
public void	setOrLimit(IloInt numberOfChoicePoints) const
public void	setPropagationControl(IloNumVar x) const
public void	setPropagationControl(IloIntVar x) const
public void	setRelativeOptimizationStep(IloNum step) const
public void	setSimpleTupleSet() const
public void	setTimeLimit(IloNum time) const
public void	setTrace(const IlcTrace trace) const
public void	setTraceMode(IloBool trace) const
public void	setValue(const IloNumVar v, IloNum n) const
public IloBool	solve() const
public IloBool	solve(const IlcGoal goal, IloBool restore=IloFalse) const
public IloBool	solve(const IloGoal goal, IloSynchronizeMode mode=IloSynchronizeAndRestart, IloBool restore=IloFalse) const
public IloBool	solveFeasible() const
public IloBool	solveFeasible(const IloGoal goal, IloSynchronizeMode mode=IloSynchronizeAndRestart, IloBool restore=IloFalse) const
public void	startNewSearch(const IlcGoal goal) const
public void	startNewSearch(const IloGoal goal=0, IloSynchronizeMode mode=IloSynchronizeAndRestart)

	const
public void	unsetLimit() const
public void	unuse(const IlcConstraintAggregator a) const
public void	use(const IlcConstraintAggregator a) const
public void	useHeap(IlcBool useIt) const
public void	useLinPropagation(IlcBool keep=IlcTrue, IloNum precision=0.1) const
public void	useNonLinConstraint(IlcBool keep=IlcTrue) const

Inherited Methods from `IloAlgorithm`
clear, end, error, extract, getEnv, getIntValue, getIntValues, getModel, getObjValue, getStatus, getTime, getValue, getValue, getValue, getValue, getValues, getValues, isExtracted, out, printTime, resetTime, setError, setOut, setWarning, solve, warning

Inner Enumeration
IloSolver::FailureStatus

Constructors

public **IloSolver**(const IloEnv & env)

This constructor creates an algorithm for IBM® ILOG® Solver.

public **IloSolver**(const IloModel & model)

This constructor creates an algorithm for IBM ILOG Solver. Actual propagation of all constraints added to the model `model` will take place either at the next `solve()` or at the next `solver.next()`.

Examples:

When you create an algorithm (an instance of `IloSolver`, for example) and extract a model for it, you can write either this line:

```
IloSolver solver(model);
```

or these two lines:

```
IloSolver solver(env);
solver.extract(model);
```

Those two lines may be useful when you want to attach your own specialized extractor to the algorithm you are creating before you extract a particular model.

Methods

public void **add**(const IlcConstraintArray constraints) const

This member function adds all the constraints in the array `constraints` to the invoking solver so that the invoking solver takes them into account during its search. You can use this member function only during search.

```
public void add(const IlcConstraint constraint) const
```

This member function adds `constraint` to the invoking solver so that the invoking solver takes `constraint` into account during its search. You can use this member function *only* during search.

```
public void addMemoryManager(IlcMemoryManagerI * mm) const
```

This member function adds the memory manager indicated by `mm` to the list of memory managers known to the invoking solver.

When you call the member function `IloSolver::end` or when the solver re-extracts the model during synchronization, it will call the virtual member function `IlcMemoryManagerI::end` for all the memory managers known to the invoking solver.

The purpose of a memory manager is to delete automatically any memory not allocated through the Solver heap. Consequently, this member function is conventionally used at run time, like this:

```
solver.addMemoryManager(new(solver.getHeap()) MyMemoryManager());
```

where `solver` is the invoking solver and `MyMemoryManager` is a class where you have defined `MyMemoryManager::end` to delete memory appropriately.

```
public void addReversibleAction(const IlcGoal goal) const
```

This member function adds the `goal` to the invoking solver. The invoking solver will then treat `goal` as a reversible object with respect to memory management, backtracking, choice points, and so forth.

It is possible to “undo” actions that are not simple assignments—such actions as drawing in a graphic interface or dealing with files.

For managing reversible changes that are not assignments, Solver provides *reversible actions*. Such actions are executed when Solver restores the state of the invoking solver. A reversible action is created by the member function `addReversibleAction`.

However, it is much more efficient to use the reversible classes—`IlcRevInt`, `IlcRevAny`, `IlcRevBool`, `IlcRevFloat`—to manage reversible assignments than to use reversible actions.

In the member function `addReversibleAction`, the argument `goal` must be a goal that does not change the state of the invoking solver. That is, it must not execute any reversible assignments, nor call other goals, nor call the member function `fail`. The member function `addReversibleAction` saves its argument, `goal`, as a reversible action. If Solver backtracks to a choice point that was set before this call to `addReversibleAction`, then `goal` will be executed.

Since reversible actions do not have subgoals, demons are usually used for implementing them.

The execution of reversible actions is interleaved with the restoration of reversible states. Thus when a reversible goal is called, the state of the invoking solver is the same as the state of the invoking solver when `addReversibleAction` was called.

Reversible actions can be used to display the search for a solution through a graphic user interface. See the *Solver User's Manual* for an example (about animating graphic interfaces) showing how to do this.

```
public void addTrace(IloCPTrace search)
```

This member function adds a hook to set the trace to the start of the search `search`.

`public IloBool **assumeStrictNumericalDivision**() const`

This member function indicates whether the arithmetic division operator `/` is restricted to numeric division only. If the return value is `IloTrue`, `operator /` is restricted, and integer division is accessible only through the IloDiv function.

If the return value is `IloFalse`, `operator /` * performs division resulting in, in general, a non-integral result.

For compatibility with previous versions of Solver, the default value is `IloFalse`, meaning the division of two integer expressions using `operator` is another integer expression.

`public void **assumeStrictNumericalDivision**(IloBool snd) const`

This member function specifies whether the arithmetic division operator `/` is restricted to numeric division only. If `snd` is `IloTrue`, `operator /` is restricted, and integer division is accessible only through the IloDiv function.

If `snd` is `IloFalse`, `operator /` performs division resulting in, in general, a non-integral result.

For compatibility with previous versions of Solver, the default value is `IloFalse`, meaning the division of two integer expressions using `operator` is another integer expression.

`public void **commitSearch**(IloAny label) const`

This member function commits the current search to be confined to the part of the search tree below the closest choice point with label `label`. In case no such choice point exists, this method then commits the search to the current state and removes all remaining open choice points or open search nodes.

`public void **end**()`

This member function "cleans house" by freeing all memory allocated by Solver on the Solver heap for the invoking solver. Since it destroys the invoking solver and all objects associated with the invoking solver, you must *not* use the solver nor any object created with this solver after you call this member function. This member function is called automatically by `IloEnv::end`.

If you also want this member function to delete automatically memory that you allocate that is *not* on the Solver heap, then you should add a memory manager, an instance of `IlcMemoryManagerI`, to the list of memory managers associated with the invoking solver. See the class `IlcMemoryManagerI` for details.

See Also: IloEnv, IlcMemoryManagerI

`public void **endSearch**() const`

This member function terminates a search and deletes the internal objects created by Solver to carry out the search (such internal objects as the search tree, choice points, goal stack, etc.)

`public void **exitSearch**() const`

This member function completely exits the search.

`public void extract(const IloModel model) const`

This member function extracts all the extractable objects added to model for the invoking algorithm. This member function discards any previously extracted model. Actual propagation of all constraints added to the model `model` will take place either at the next `solve()` or at the next `solver.next()`.

`public void fail(IlcAny label=0) const`

The failure of a goal is triggered by this member function. In fact, this member function triggers the failure of the current goal. The execution of the current goal then stops, and execution returns to the last choice point satisfying the following conditions:

- The choice point has at least one remaining untried subgoal.
- The choice point was set after the current call to the member function IloSolver::solve or IloSolver::next.
- If the argument `label` is not 0 (zero), the choice point has the same label as `label`.

If no such choice point exists, then the current call to IloSolver::solve or IloSolver::next terminates and returns `IloFalse`.

If `fail` is called outside a call to IloSolver::solve or a call to IloSolver::next, then an error is raised.

In other words, when the member function `fail` is called, goal execution resumes at the last choice point with untried subgoals. It is possible to resume goal execution at an earlier choice point by associating labels with choice points. Then the member function `fail` can be called with a label, and in that case, goal execution resumes at the last choice point with that label.

`public IlcDemon getActiveDemon() const`

This member function returns the demon currently executing in the invoking solver. It returns an empty handle if there is no such demon.

`public IlcGoal getActiveGoal() const`

This member function returns the search goal currently executing in the invoking solver. It returns an empty handle if there is no such goal.

`public IlcAnyArray getAnyArray(const IloAnyArray arg) const`

This member function returns the `Ilc` class (that is, the extracted search class) corresponding to the `Ilo` class (that is, the model class) of `arg`.

`public IlcAnySet getAnySet(const IloAnySet arg) const`

This member function returns the `Ilc` class (that is, the extracted search class) corresponding to the `Ilo` class (that is, the model class) of `arg`.

`public IloAnySet getAnySetValue(const IloAnySetVar var) const`

This member function returns the value that `var` assumes in a solution.

```
public IlcAnySetVar getAnySetVar(const IloAnySetVar var) const
```

This member function returns the algorithmically constrained enumerated set variable corresponding to the modeling variable indicated by `var`.

```
public IlcAnySetVarArray getAnySetVarArray(const IloAnySetVarArray vars) const
```

This member function returns the array of algorithmically constrained enumerated set variables corresponding to the array of modeling variables indicated by `vars`.

```
public IloAny getAnyValue(const IloAnyVar var) const
```

This member function returns the modeling value that `var` assumes in a solution.

```
public IlcAnyVar getAnyVar(const IloAnyVar var) const
```

This member function returns the algorithmically constrained enumerated variable corresponding to the modeling variable indicated by `var`.

```
public IlcAnyVarArray getAnyVarArray(const IloAnyVarArray vars) const
```

This member function returns the array of algorithmically constrained enumerated variables corresponding to the array of modeling variables indicated by `vars`.

```
public IlcBox getBox(const IloBox) const
```

This member function returns the box associated with the `IloBox` object in the invoking solver.

```
public IloNum getBranchImpact(const IloNumVar x) const
```

This member function returns a floating point number between 0 and 1, which represents the proportion of removed intervals from the intervals of `x` during the last constraint propagation.

```
public IloNum getBranchImpact(const IlcFloatVar x) const
```

This member function returns a floating point number between 0 and 1, which represents the proportion of removed intervals from the intervals of `x` during the last constraint propagation.

```
public IloNum getBranchImpact(const IloIntVar x) const
```

This member function returns a floating point number between 0 and 1, which represents the proportion of removed values from the domain of `x` during the last constraint propagation.

`public IloNum getBranchImpact(const IlcIntVar x) const`

This member function returns a floating point number between 0 and 1, which represents the proportion of removed values from the domain of `x` during the last constraint propagation.

`public IlcConstraint getConstraint(const IloConstraint ct) const`

This member function returns the constraint corresponding to the modeling constraint indicated by `ct`.

`public IlcInt getCurrentNumberOfNodes() const`

This member function returns the number of active nodes currently open in the search carried on by the invoking solver.

`public IlcFloat getDefaultPrecision() const`

This member function returns the default precision of the invoking solver. That value is used in managing constrained floating-point expressions (instances of `IlcFloatExp` and its subclasses) if you associate no other relative precision with a given expression. That value in a solver is always greater than or equal to 2.10-11. It is initially 1.10-10

`public IloInt getDegree(const IloIntVar x) const`

This member function returns the degree of the variable `x`. The degree is the number of uninstantiated variables appearing in a constraint where `x` also appears. This value changes throughout search. This function can be called only when the aggregator obtained by the function `IlcDegreeInformation` is used. Moreover, it can be called only on variables corresponding to `IloIntVar` appearing in the extracted model.

`public IloInt getDegree(const IlcIntVar x) const`

This member function returns the degree of the variable `x`. The degree is the number of uninstantiated variables appearing in a constraint where `x` also appears. This value changes throughout search. This function can be called only when the aggregator obtained by the function `IlcDegreeInformation` is used. Moreover, it can be called only on variables corresponding to `IloIntVar` appearing in the extracted model.

`public IlcFloat getElapsedTime() const`

This member function displays part of the statistics about the current state of the invoking solver available from `IloSolver::printInformation`. In particular, it returns the the elapsed time (sometimes known as wall clock time) of the solver, in seconds, since the most recent extraction of a model or the most recent synchronization of a model.

`public IloSolver::FailureStatus getFailureStatus() const`

This member function returns an identifier (one of the choices of the nested enumeration `IloSolver::FailureStatus`) that indicates the failure status of an instance of `IloSolver` after an unsuccessful IloSolver::solve or IloSolver::next. Possible values of `FailureStatus` are:

```
enum FailureStatus {
                    searchHasNotFailed,
                    searchFailedNormally,
                    searchStoppedByLimit,
                    searchStoppedByLabel,
                    searchStoppedByExit,
                    unknownFailureStatus
};
```

`public IlcFloatArray` **`getFloatArray`**`(const IloNumArray arg) const`

This member function returns the `Ilc` class (that is, the extracted search class) corresponding to the `Ilo` class (that is, the model class) of `arg`.

`public IlcFloatDisplay` **`getFloatDisplay`**`() const`

This member function returns a value (one of the choices of the enumeration `IlcFloatDisplay`) that indicates how the invoking solver will display the values of constrained floating-point variables as output.

`public IlcFloatExp` **`getFloatExp`**`(const IloNumExprArg expr) const`

This member function returns the floatting point expression corresponding to the modeling numerical expression indicated by `expr`.

`public IlcFloatVar` **`getFloatVar`**`(const IloNumVar var) const`

This member function returns the algorithmically constrained floating-point variable corresponding to the modeling variable indicated by `var`.

`public IlcFloatVarArray` **`getFloatVarArray`**`(const IloNumVarArray vars) const`

This member function returns the array of algorithmically constrained floating-point variables corresponding to the modeling variables indicated by `vars`.

`public IlcAllocationStack *` **`getHeap`**`() const`

This member function returns a pointer to the heap associated with the invoking solver. Use this member function with the overloaded `new` operator, like this `new (solver.getHeap())`.

`public IlcFloat` **`getImpact`**`(const IlcIntVar x) const`

This member function returns the impact of a variable. This impact is the sum of the impact of each value in its current domain.

`public IlcFloat` **`getImpact`**`(const IlcIntVar x, IlcInt v, IlcBool countFails=IlcTrue) const`

This member function returns the average of observed impacts on the assignment `x = v` so far. The impact of an assignment `x = v` is the proportion of the search space that this assignment eliminates by constraint propagation. When `countFails` is set to `IloTrue`, failures of the assignment are also counted as having an impact of 1.0. Otherwise they are not taken into account in the average.

`public IloSolverI * getImpl() const`

This member function returns the undocumented class `IloSolverI*`.

`public IlcIntArray getIntArray(const IloNumArray arg) const`

This member function returns the `Ilc` class (that is, the extracted search class) corresponding to the `Ilo` class (that is, the model class) of `arg`.

`public IlcIntExp getIntExp(const IloIntExprArg expr) const`

This member function returns the integer expression corresponding to the modeling integer expression indicated by `expr`.

`public IlcIntSet getIntSet(const IloIntSet arg) const`

This member function returns the `Ilc` class (that is, the extracted search class) corresponding to the `Ilo` class (that is, the model class) of `arg`.

`public IloIntSet getIntSetValue(const IloIntSetVar var) const`

This member function returns the modeling value that `var` assumes in a solution.

`public IlcIntSetVar getIntSetVar(const IloIntSetVar var) const`

This member function returns the algorithmically constrained set variable corresponding to the modeling variable indicated by `var`.

`public IlcIntSetVarArray getIntSetVarArray(const IloNumSetVarArray vars) const`

This member function returns the array of algorithmically constrained integer set variables corresponding to the array of modeling variables indicated by `vars`.

`public IlcIntVar getIntVar(const IloNumVar var) const`

This member function returns the algorithmically constrained integer variable corresponding to the modeling variable indicated by `var`.

`public IlcIntVarArray` **`getIntVarArray`**`(const IloNumVarArray vars) const`

This member function returns the array of algorithmically constrained integer variables corresponding to the modeling variables indicated by `vars`.

`public IlcFloat` **`getLocalImpact`**`(const IlcIntVar x, IlcInt v) const`

This member function computes and returns the impact on the assignment `x = v`. The impact is the proportion of the search space that this assignment eliminates by constraint propagation at the place where this function is called.

`public IlcFloat` **`getLocalVarImpact`**`(const IlcIntVar x, IlcInt depth=-1) const`

This member function computes the impact of the assignment `x = v` for each value `v` in the current domain of `x` and returns the sum of these impacts.

`public IloNum` **`getMax`**`(const IloNumVar v) const`

This member function returns the maximum value of the numeric variable `v`.

`public IlcUInt` **`getMaxSearchMemoryUsage`**`() const`

This member function returns the maximum amount of memory used to store nodes during the search by the invoking solver.

`public IlcUInt` **`getMemoryUsage`**`() const`

This member function displays part of the statistics about the current state of the invoking solver available from `IloSolver::printInformation`. In particular, it returns the total memory, in bytes, used by the invoking solver.

`public IloNum` **`getMin`**`(const IloNumVar v) const`

This member function returns the minimum value of the numeric variable `v`.

`public IlcFloat` **`getNonLinPrecision`**`() const`

When you are using non-linear constraint propagation (by means of the member function `IloSolver::useNonLinConstraint`), this member function returns the precision of non-linear constraint propagation with respect to floating-point values and with respect to the safety of solutions.

`public IlcInt` **`getNumberOfChoicePoints`**`() const`

This member function displays part of the statistics about the current state of the invoking solver available from `IloSolver::printInformation`. In particular, it returns the number of choice points since the most recent extraction or synchronization of a model by the invoking solver.

`public IlcInt` **`getNumberOfConstraints`**`() const`

This member function displays part of the statistics about the current state of the invoking solver available from `IloSolver::printInformation`. In particular, it returns the number of constraints extracted for the invoking solver.

`public IlcInt` **`getNumberOfCuts`**`() const`

This member function returns the number of nodes discarded during the search by the invoking solver.

`public IlcFloat` **`getNumberOfFails`**`(const IlcIntVar x, IlcInt v) const`

This member function returns the number of times the assignment `x = v` has failed at this point in the search.

`public IlcInt` **`getNumberOfFails`**`() const`

This member function displays part of the statistics about the current state of the invoking solver available from `IloSolver::printInformation`. In particular, it returns the number of failures since the most recent extraction or synchronization of a model by the invoking solver.

A reset of a model in the invoking solver will occur under any of these conditions:

- You explicitly extract a new model into the solver.
- You synchronize the invoking solver after changing the model during a search.
- You synchronize the invoking solver by calling the member function `startNewSearch(IlcSynchronizeAndRestart).`

`public IlcInt` **`getNumberOfFiles`**`() const`

This member function returns the number of node files used during the search by the invoking solver.

`public IlcFloat` **`getNumberOfInstantiations`**`(const IlcIntVar x, IlcInt v) const`

This member function returns the number of times the assignment `x = v` has been made at this point in the search.

`public IlcInt` **`getNumberOfMoves`**`() const`

This member function returns the number of moves performed during the search in the search tree by the invoking solver since the most recent extraction or synchronization of a model by the invoking solver.

`public IlcInt` **`getNumberOfRecomputed`**`() const`

This member function returns the number of choice points recomputed by the invoking solver since the most recent extraction or synchronization of a model by the invoking solver.

`public IlcInt getNumberOfVariables() const`

This member function displays part of the statistics about the current state of the invoking solver available from `IloSolver::printInformation`. In particular, it returns the number of constrained variables extracted for the invoking solver.

`public IloNum getOptimizationStep() const`

This member function returns the current optimization step. The optimization step indicates how much improvement there must be between solutions during the search launched by `IloSolver::solve`.

`public IlcAllocationStack * getPersistentHeap() const`

This member function returns a pointer to the heap where persistent objects of the search are stored. When the invoking solver is deleted, this heap will be deleted, too.

`public IlcRandom getRandom() const`

This member function returns a random number generator. This can be convenient as it can be reduce the number of places that a random number generator needs to be explicitly passed in user code.

`public IloNum getReduction(const IloNumVar x) const`

This member function returns the difference between the domain size of `x` before and after the the last constraint propagation.

`public IloNum getReduction(const IlcFloatVar x) const`

This member function returns the difference between the domain size of `x` before and after the last constraint propagation.

`public IloInt getReduction(const IloIntVar x) const`

This member function returns the number of removed values from the domain of `x` during the last constraint propagation.

`public IloInt getReduction(const IlcIntVar x) const`

This member function returns the number of removed values from the domain of `x` during the last constraint propagation.

`public IloNum getRelativeOptimizationStep() const`

This member function returns the relative optimization step. For example, if the optimization step is 90%, Solver searches for a solution where the objective value is less than 90% of the objective value of the best solution found so far.

`public IlcUInt` **`getSearchMemoryUsage`**`() const`

This member function returns the amount of memory currently used to store nodes by the invoking solver.

`public IlcSearchNode` **`getSearchNode`**`() const`

This member function returns the current open node being explored by the invoking solver.

`public IlcFloat` **`getSuccessRate`**`(const IlcIntVar x) const`

This member function returns the rate of instantiations `x = v` that fails by considering only values `v` that are in the current domain of `x`.

`public IlcFloat` **`getSuccessRate`**`(const IlcIntVar x, IlcInt v) const`

This member function returns the success rate of assignment `x = v` at this point in the search. The success rate is the proportion of assignments of this kind that does not fail.

`public IlcFloat` **`getTime`**`() const`

This member function displays part of the statistics about the current state of the invoking solver available from `IloSolver::printInformation`. In particular, it returns the total running time of the solver, in seconds, since the most recent extraction of a model or the most recent synchronization of a model.

`public IlcInt` **`getTotalNumberOfNodes`**`() const`

This member function returns the total number of nodes created during the search by the invoking solver since the most recent extraction or synchronization of a model by the invoking solver.

`public IloNum` **`getValue`**`(const IloNumVar v) const`

This member function returns the value of the numeric variable `v`.

`public IlcInt` **`getWorkerId`**`() const`

This member function returns a positive integer identifying the current worker (an instance of `IloSolver`). These identifying numbers start from `0` (zero) and are contiguous. In a sequential search, this member function always returns `0`.

`public IloBool **isExtracted**(const IloExtractable ext) const`

This member function checks whether an extractable `ext` has been extracted by Solver.

`public IloBool **isInRecomputeMode**() const`

This member function returns `IloTrue` if the invoking solver is jumping from one open node to another using recomputation. It returns `IloFalse` otherwise.

`public IloBool **isInSearch**() const`

This member function returns `IloTrue` if the invoking solver is conducting a search; otherwise, it returns `IloFalse`.

`public IloBool **isInteger**(const IloNumVar var) const`

This member function returns `IloTrue` if var represents an integer variable in the invoking solver; it returns `IloFalse` otherwise.

`public IloBool **isNonLinSafe**() const`

This member function applies only to the non linear equalities and inequalities in a model; it says nothing about constraints that are *not* non linear equalities or inequalities. It returns `IloTrue` if the solution returned by the invoking solver has been proved safe in the sense explained in *Safe Solutions*. In short, there is a solution (an assignment of values to variables satisfying all non linear constraints in the model) in the interval returned by the solver. It returns `IloFalse` otherwise.

`public IloBool **next**() const`

This member function searches for the next solution in the search tree of the invoking `IloSolver` object.

The first time `next` is called, it iteratively pops one goal from the goal stack and executes it. The execution of the goal can add other goals to the stack and can set choice points.

The execution of `next` terminates in two cases. First, when the goal stack becomes empty, this member function returns `IloTrue`. Second, if a failure occurs and no choice point with untried subgoals and correct labels exists, the member function restores the state of the invoking search and returns `IloFalse`.

The second execution and subsequent executions of `next` start from where the preceding execution left off. If a solution was found, then Solver backtracks and searches for the next solution. If no solution was found, this member function immediately returns `IloFalse`.

This member function returns `IloTrue` when it has found a solution.

`public void **printInformation**(ostream & stream) const`

This member function is part of the special purpose debugging features of Solver. It displays statistics about the current state of the solver since the most recent extraction of a model or the most recent synchronization of a model and prints to stream `stream`. For more information, see `IloSolver::printInformation`.

```
public void printInformation() const
```

This member function is part of the special purpose debugging features of Solver. It displays statistics about the current state of the solver since the most recent extraction of a model or the most recent synchronization of a model.

Specifically, in the output stream indicated by its argument, this member function displays the following information about memory use by Solver:

- the number of failures;
- the number of choice points;
- the number of constrained variables;
- the number of posted constraints to the invoking solver;
- the size of the restoration stack; (This size is proportional to the maximal number of reversible assignments.)
- the maximal size of the Solver allocation heap;
- the size of the goal stack; (This number is proportional to the number of calls to the function IlcAnd.)
- the size of the IlcOr stack; (This number is proportional to the number of calls to the function IlcOr.)
- the size of the constraint propagation queue; (This number is proportional to the number of activated constraints.)
- the total amount of memory used by Solver for the invoking solver;
- the total running time.

Each of those numbers can be accessed individually by one of the following member functions.

- number of failures - getNumberOfFails
- number of choice points - getNumberOfChoicePoints
- number of constrained variables - getNumberOfVariables
- number of posted constraints - getNumberOfConstraints
- amount of memory used - getMemoryUsage
- total running time - getTime

Typical output from the member function printInformation looks like this:

```
Number of fails              : 3306
Number of choice points      : 3310
Number of variables          : 26
Number of constraints        : 21
Reversible stack (bytes)     : 8060
Solver heap (bytes)          : 16100
And stack (bytes)            : 4040
Or stack (bytes)             : 4040
Constraint queue (bytes)     : 4068
Total memory used (bytes)    : 36308
Total CPU time               : 12.29
```

```
public IloBool propagate(const IloConstraint constraint=0, IloSynchronizeMode
mode=IloSynchronizeAndRestart, IloBool restore=IloFalse) const
```

This member function propagates the constraint `constraint` and synchronizes with the model (or not) according to the parameter `mode`. This member functions returns `IloTrue` if the propagation succeeds; otherwise, it returns `IloFalse`. The parameter `restore` indicates whether or not at the end of the propagation the invoking solver should restore the state that it was in prior to the search. This restoration of a state influences only the propagation, not the synchronization with a model. The invoking solver will remain synchronized (or not, consistent with the `mode` parameter) after the search regardless of the `restore` parameter. This member function must be used at the top level of the search.

`public void **removeTrace**(IloCPTrace search)`

This member function removes a hook to set the trace from the search `search`.

`public void **restartSearch**() const`

This member function restarts the search at the top of the search tree (that is, before the first call of the member function `next`).

It also stores the best value of any objective function that has been defined by `IloSolver::setObjMin`.

If you have stored an intermediate solution, then you should call `restart` before you call `IloSolver::next` so that `next` will immediately produce that stored solution.

`public void **setBigTupleSet**() const`

This member function controls how Solver manages the propagation of tuple set table constraints. It must be called before extraction. It allows you to modify the default behavior (set using `IloSolver::setSimpleTupleSet`) to accelerate the propagation of table constraints defined by the set of their allowed tuples when propagation is slow. Typically, this can occur when the tuples are of an arity greater than three and when they contain numerous tuples. `IloSolver::setBigTupleSet` uses a data structure that is very quick, but that consumes a lot of memory.

`public void **setDefaultFilterLevel**(IlcFilterLevelConstraint ct, IlcFilterLevel level) const`

This member function defines the default filter level for a *type* of constraint, such as the types `IlcAllDiff`, `IlcDistribute`, `IlcSequence`, and so forth. For each type of constraint (as defined in the enumeration `IlcFilterLevelConstraint`) Solver respects a default filter level in propagation. With this member function, you can reset that default for all constraints of that type.

If you want to reset the default filter level of a given constraint (rather than for all constraints of a given type), then consider the member function `IloSolver::setFilterLevel`.

`public void **setDefaultPrecision**(IloNum precision) const`

This member function sets the default precision of the invoking solver. That value is used in managing constrained floating-point expressions (instances of `IlcFloatExp` and its subclasses) if you associate no other relative precision with a given expression. In any case, the default precision must be greater than or equal to 2.10-11. When it is less than 2.10-11, the default precision is automatically changed to 2.10-11. This member function is *not* reversible. This member function does *not* change the precision of any constrained floating-point expressions that have already been created.

`public void **setFailLimit**(IloInt numberOfFailures) const`

This member function sets a limit on the number of failures during a search performed by `IloSolver::solve` or by `IloSolver::startNewSearch`.

The limit is set to the current number of fails plus `numberOfFailures`. The limit is recomputed each time this member function is called. When the limit is reached, the search stops and the current call to the member function `IloSolver::solve` returns `IloFalse`.

The effect of this member function is immediate. This change will influence any subsequent calls of the member functions `IloSolver::startNewSearch` and `IloSolver::solve`.

`public void` **`setFastRestartMode`**`(IloBool mode) const`

This member function controls initial constraint propagation during a search performed by `IloSolver::startNewSearch`. The initial constraint propagation will not be redone if the following conditions are met:

- synchronization mode has been set to `IloSynchronizeAndRestart`
- the model has not changed since the last search
- the parameter `mode` is `IloTrue`

This mode is not the default.

`public void` **`setFileNodeOptions`**`(IlcInt maxSize, char * prefixName, IlcBool useCompression, IlcBool useDisk) const`

Node files make it possible for you to limit the amount of memory Solver uses to store open search nodes. (Open search nodes in the search are the ones which have not yet been completely explored.) You activate node files by invoking this member function. This member function sets the options for node files and must be used before starting a search.

When the memory used to store nodes is greater than `maxSize` bytes, Solver creates a buffer of one megabyte. Solver then fills that buffer with open nodes. The parameter `maxSize` cannot be less than 5 000 000. If the given parameter is less than 5 000 000, Solver will silently change it to 5 000 000.

If the parameter `useCompression` is set to `IlcTrue`, the buffer is compressed. If the parameter `useDisk` is set to `IlcTrue`, this temporary buffer is then flushed from memory and written to disk as a file. The name of the file is prefixed by `prefixName`.

If one buffer is not enough to reduce the memory consumption below `maxSize`, then Solver creates node files until the memory consumption fits that limit.

`public void` **`setFilterLevel`**`(IlcConstraint ct, IlcFilterLevel level) const`

This member function defines the filter level for a given constraint. For each type of constraint (as defined in the enumeration `IlcFilterLevelConstraint`) Solver respects a default filter level in propagation. With this member function, you can reset the filter level of a given constraint to override the default filter level of its type.

If you want to reset the default filter level of all constraints of a given type, then consider the member function `setDefaultFilterLevel`.

`public void` **`setFilterLevel`**`(IloConstraint oct, IlcFilterLevel level) const`

This member function defines the filter level for a given constraint. For each type of constraint (as defined in the enumeration `IlcFilterLevelConstraint`) Solver respects a default filter level in propagation. With this member function, you can reset the filter level of a given constraint to override the default filter level of its type.

If you want to reset the default filter level of all constraints of a given type, then consider the member function `setDefaultFilterLevel`.

`public void setFloatDisplay(IlcFloatDisplay display) const`

This member function changes the format of the display of floating-point constrained variables. Possible values of `display` are:

```
enum IlcFloatDisplay {
    IlcStandardDisplay = 0,
    IlcIntScientific   = 1,
    IlcIntFixed        = 2,
    IlcBasScientific   = 3,
    IlcBasFixed        = 4
};
```

The default value is `IlcIntFixed`. See the enumeration `IlcFloatDisplay` for an explanation of those values.

`public void setHoloTupleSet() const`

This member function controls how Solver manages the propagation of tuple set table constraints. It must be called before extraction. It allows you to modify the default behavior (set using `IloSolver::setSimpleTupleSet`) to accelerate the propagation of table constraints defined by the set of their allowed tuples when propagation is slow. Typically, this can occur when the tuples are of an arity greater than three and when they contain numerous tuples. `IloSolver::setHoloTupleSet` uses a data structure that is almost as quick as that used by `IloSolver::setBigTupleSet`, but that only consumes slightly more memory than the default `IloSolver::setSimpleTupleSet` data structure.

`public void setMax(const IloNumVar v, IloNum n) const`

This member function assigns `n` as the maximum value of the numeric variable `v`.

`public void setMin(const IloNumVar v, IloNum n) const`

This member function assigns `n` as the minimum value of the numeric variable `v`.

`public void setMonitor(const IlcSearchMonitor monitor, IloBool rec) const`

This member function assigns `mon` as the search monitor for the invoking search. A monitor watches and reports events in a search. If the `IloBool` is `IlcTrue` this monitor is set to monitor all searches including nested searches. If the `IloBool` is `IlcFalse` the monitor is set only for the top level search. This member function must be used before starting a search.

`public void setNonLinPrecision(IlcFloat p) const`

This member function sets `p` as the degree of precision to use with nonlinear propagation techniques (see the member function `IloSolver::useNonLinConstraint`) and in determining the safety of nonlinear solutions (see the member function `IloSolver::isNonLinSafe`). By default, the nonlinear precision is equal to the default precision of the invoking solver.

`public void setObjMin(const IlcFloatVar & obj, IloNum step, IloNum r=0.0) const`

This member functions sets an optimization objective to minimize in the invoking search. There is only one objective at a time for a given search. In other words, each time you call this member function, it replaces the previous objective (if there was one). Specifically, this member function sets `obj` as the minimum objective of the invoking search object and sets `step` as the step size. That is, it constrains the search to produce solutions that are at least a step better than the previous solution.

If you want to maximize an objective, then use `solver.setObjMin(-obj, step);`.

`public void` **`setObjMin`**`(const IlcIntVar & obj, IloInt step=1, IloNum r=0.0) const`

This member function sets an optimization objective to minimize in the invoking search. There is only one objective at a time for a given search. In other words, each time you call this member function, it replaces the previous objective (if there was one). Specifically, this member function sets `obj` as the minimum objective of the invoking search object and sets `step` as the step size. That is, it constrains the search to produce solutions that are at least a step better than the previous solution.

If you want to maximize an objective, then use `solver.setObjMin(-obj, step);`.

`public void` **`setOptimizationStep`**`(IloNum step) const`

This member function sets the step size used to measure improvement between solutions during a search by `IloSolver::solve`.

You can also set the step size for a given *variable* by means of a parameter in the function `IloMinimizeVar`.

`public void` **`setOrLimit`**`(IloInt numberOfChoicePoints) const`

This member functions sets a limit on the number of choice points (corresponding to execution of the goal `IlcOr`) during a search by `IloSolver::solve` or by `IloSolver::startNewSearch`.

The limit is set to the current number of choice points plus `numberOfChoicePoints`. The limit is recomputed each time this member function is called. When the limit is reached, the search stops and the current call to the member function `IloSolver::solve` returns `IloFalse`.

The effect of this member function is immediate. This change will influence any subsequent calls of the member functions `IloSolver::startNewSearch` and `IloSolver::solve`.

`public void` **`setPropagationControl`**`(IloNumVar x) const`

This member function allows you to control propagation for variables that you suspect have become stuck in a propagation cycle, resulting sometimes in very slow constraint propagation. For example, the domain of variable x is reduced. This reduction causes the reduction of the domain of variable y, which causes the reduction of the domain of variable z, and so on, until this cycle of reductions returns to reduce the domain of variable x. This function must be called before extraction.

`public void` **`setPropagationControl`**`(IloIntVar x) const`

This member function allows you to control propagation for variables that you suspect have become stuck in a propagation cycle, resulting sometimes in very slow constraint propagation. For example, the domain of variable x is reduced. This reduction causes the reduction of the domain of variable y, which causes the reduction of the domain of variable z, and so on, until this cycle of reductions returns to reduce the domain of variable x. This function must be called before extraction.

```
public void setRelativeOptimizationStep(IloNum step) const
```

This member function sets the relative step size used to measure improvement between solutions during a search by `IloSolver::solve`. It is used to update the cutoff each time a mixed integer solution is found. The value is multiplied by the absolute value of the integer objective and subtracted from (added to) the newly found integer objective when minimizing (maximizing). This forces the mixed integer optimization to ignore integer solutions that are not at least this amount better than the one found so far. This value must be strictly between 0 and 1.

```
public void setSimpleTupleSet() const
```

This member function controls how Solver manages the propagation of tuple set table constraints. It must be called before extraction. This is the default setting for tuple set constraint propagation. The member functions `IloSolver::setBigTupleSet` and `IloSolver::setHoloTupleSet` allow you to modify this default behavior to accelerate the propagation of table constraints defined by the set of their allowed tuples when propagation is slow. Typically, this can occur when the tuples are of arity greater than three and when they contain numerous tuples.

```
public void setTimeLimit(IloNum time) const
```

This member function sets a limit on the amount of time spent during a search by `IloSolver::solve` or by `IloSolver::startNewSearch`.

The limit is set to the current time plus `time`. The limit is recomputed whenever this member function is called. When the limit is reached, the search stops and the current call to the member function `IloSolver::solve` returns `IlcFalse`.

The time is measured in elapsed CPU seconds for the search process.

This change will influence any subsequent calls of the member functions `IloSolver::startNewSearch` and `IloSolver::solve`.

```
public void setTrace(const IlcTrace trace) const
```

This member function assigns `trace` as the trace associated with the invoking solver.

```
public void setTraceMode(IloBool trace) const
```

This member function turns the trace mechanism on or off for the invoking solver. If the value of trace is `IlcTrue`, the trace mechanism turns on. If the value of trace is `IlcFalse`, the trace mechanism turns off.

```
public void setValue(const IloNumVar v, IloNum n) const
```

This member function assigns `n` as the value of the numeric variable `v`.

```
public IloBool solve() const
```

This member function solves a problem by using a default goal to launch the search. It is used only at the top level of the search.

This member function first checks to see whether a model has already been extracted. If a model has already been extracted, it then checks whether the model has changed since it was extracted previously. If the model has changed, then the model is extracted again. If the model has not changed, then it is not extracted again. In other words, this member function synchronizes with the current model before it starts its search.

When Concert Technology determines at extraction time that the model is infeasible, it skips the remainder of the extraction, and the first call to the member function `solve` will report *without any search* that there is no solution. If an instance of the class `IloObjective` has been added to the model, then the solver will search for an optimal solution.

`public IloBool` **`solve`**`(const IlcGoal goal, IloBool restore=IloFalse) const`

This member function solves a problem by using the goal `goal` passed as a parameter. The parameter `restore` indicates whether or not at the end of the search the invoking solver should restore the state that it was in prior to the search. This member function can only be used within a search (for example, inside a goal) not at the top level.

`public IloBool` **`solve`**`(const IloGoal goal, IloSynchronizeMode mode=IloSynchronizeAndRestart, IloBool restore=IloFalse) const`

This member function solves a problem by using the `goal` passed as a parameter. By means of the `mode` parameter, you control how solver synchronizes with the current model. When using the default parameter, `IloSynchronizeAndRestart`, the instance of `IloSolver` will end in a state equivalent to a deletion of the instance of `IloSolver` and a reload of the current model. Use the parameter `IloSynchronizeAndContinue` only if all changes since the last synchronization are monotonic. In this case, the instance of `IloSolver` will apply all the changes from the current state and continue from there. The parameter `restore` indicates whether or not at the end of the search the invoking solver should restore the state that it was in prior to the search. This restoration of a state influences only the search, not the synchronization with a model. This member function is used only at the top level of the search.

`public IloBool` **`solveFeasible`**`() const`

This member function behaves exactly like IloSolver::solve, except in the case where the model contains an objective (an instance of IloObjective). In this case, `solveFeasible` finds the first solution respecting all the problem constraints, whereas IloSolver::solve finds a solution respecting all constraints that also optimizes the objective.

`public IloBool` **`solveFeasible`**`(const IloGoal goal, IloSynchronizeMode mode=IloSynchronizeAndRestart, IloBool restore=IloFalse) const`

This member function behaves exactly like IloSolver::solve, except in the case where the model contains an objective (an instance of IloObjective). In this case, `solveFeasible` finds the first solution respecting all the problem constraints, whereas IloSolver::solve finds a solution respecting all constraints that also optimizes the objective.

`public void` **`startNewSearch`**`(const IlcGoal goal) const`

This member function starts a new search with `goal.` This function is only used inside a search.

`public void startNewSearch(const IloGoal goal=0, IloSynchronizeMode mode=IloSynchronizeAndRestart) const`

This member function starts a new search with `goal` and synchronizes with the model (or not) according to the parameter `mode`. This member function is used only at the top level of the search.

`public void unsetLimit() const`

This member function removes any limits, such as a time limit, a limit on the number of choice points, or a limit on the number of failures, for the invoking solver. This change will influence any subsequent calls of the member functions `IloSolver::startNewSearch` and `IloSolver::solve`.

`public void unuse(const IlcConstraintAggregator a) const`

This member function removes the constraint aggregator `agg` from the current list of aggregators in the invoking object.

`public void use(const IlcConstraintAggregator a) const`

This member function adds the constraint aggregator `agg` to the current list of aggregators in the invoking object. This function must be called before model extraction.

`public void useHeap(IlcBool useIt) const`

If the value of `useIt` is `IlcTrue`, then this member function makes the invoking solver use the overloaded `new` operator to create objects associated with the invoking solver. Those objects are then allocated on the heap associated with that solver; they will be de-allocated when you call the member function `end` for that solver.

If the value of `use` is `IlcFalse`, then Concert Technology uses the conventional C++ `new` operator.

`public void useLinPropagation(IlcBool keep=IlcTrue, IloNum precision=0.1) const`

This member function changes how Solver handles the propagation of linear constraints. After this member function is called, all linear constraints added to Solver are no longer handled by the default propagation mechanism of Solver. Instead they are collected by a global propagation mechanism. This global propagation can improve performance in cases where the linear constraints reduce the variable domains only by small increments.

```
int main () {
  IloEnv env;
  IloModel model(env);
  IloSolver solver(model);
  IloIntVar var1(env, 0, 20000000);
  IloIntVar var2(env, 0, 20000000);
  model.add(var1 < var2);
  model.add(var1 > var2);

  solver.useLinPropagation();

  if(solver.propagate()) {
    cout << "Propagation succeeded" << endl;
    solver.printInformation();
  }
```

```
  else {
    cout << "Propagation failed" << endl;
    solver.printInformation();
  }
  env.end();
  return 0;
}
```

On this example, the use of `useLinPropagation` significantly improves the running time. Note, however, that the aforementioned global propagation mechanism is more costly and hence not always beneficial with regard to the default propagation mechanism of Solver.

`public void` **`useNonLinConstraint`**`(IlcBool keep=IlcTrue) const`

This member function tells the invoking solver to use nonlinear propagation techniques. In order to use these techniques, you must call this member function before you extract your model for the invoking solver. For example,

```
solve.useNonLinConstraint();
```

Then for all the nonlinear constraints in your model, the invoking solver will use nonlinear propagation techniques.

In addition, when the set of nonlinear constraints is a square system of equations or a square system of inequalities, a global constraint will be posted. This constraint may allow for additional propagation.

Furthermore it may allow you to prove that the solution-interval found for the system of nonlinear constraints surely contains a solution.

The `keep` parameter specifies whether or not in addition to the advanced propagation techniques the invoking solver should also use its default propagation. If `keep` is `IlcTrue` constraints are propagated only by the nonlinear propagation techniques; if `keep` is `IlcFalse`, the invoking solver will use nonlinear propagation techniques as well as its default propagation techniques.

The default value of `keep` is `IlcTrue`.

Inner Enumerations

Enumeration FailureStatus

Definition file: ilsolver/ilosolverhandle.h
Include file: <ilsolver/ilosolver.h>

The values in this enumeration indicate the failure status of an instance of `IloSolver` after an unsuccessful IloSolver::solve or IloSolver::next. The member function IloSolver::getFailureStatus is used to query the failure status of an instance of `IloSolver`.

See Also: IloSolver

Fields:

`searchHasNotFailed`

`searchFailedNormally`

`searchStoppedByLimit`

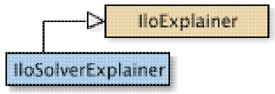
`searchStoppedByLabel`

searchStoppedByExit

unknownFailureStatus

Class IloSolverExplainer

Definition file: ilsolver/iloexplain.h
Include file: <ilsolver/iloexplain.h>



Solver now provides the ability to explain why a particular solution has been proposed. An instance of the class `IloSolverExplainer` is an explainer that uses the propagation mechanism of an IloSolver for deducing queries and fails.

> **Note**
>
> This functionality is available for pure Solver code only. It will not work on IBM® ILOG® Dispatcher or IBM ILOG Scheduler code.

Constructor Summary	
`public`	`IloSolverExplainer(IloSolverExplainerI * impl=0)`
`public`	`IloSolverExplainer(IloModel model, IloBool deleteQuery=IloFalse)`

Method Summary	
`public IloSolverExplainerI *`	`getImpl() const`

Inherited Methods from `IloExplainer`
`end, getImpl, why, whyFail, whyNot`

Constructors

`public` **`IloSolverExplainer`**`(IloSolverExplainerI * impl=0)`

This constructor creates a handle of a Solver explainer with the implementation object `impl`. The class `IloSolverExplainerI` is not documented.

`public` **`IloSolverExplainer`**`(IloModel model, IloBool deleteQuery=IloFalse)`

This constructor creates a Solver explainer for the model `model`. The explanations found by this explainer are subsets of the set of constraints of model.

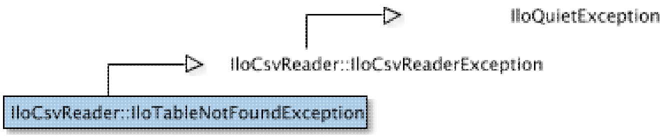
Methods

`public IloSolverExplainerI *` **`getImpl`**`() const`

This member function returns the implementation object of the invoking explanation object.

Class IloCsvReader::IloTableNotFoundException

Definition file: ilconcert/ilocsvreader.h

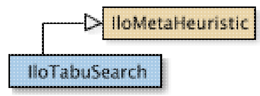


Exception thrown for unfound table.
This exception is thrown by the constructor `IloCsvTableReader(IloCsvReaderI *, const char * name = 0)` and by the member functions listed below if the table you want to construct or to get is not found.

- IloCsvReader::getTableByNumber
- IloCsvReader::getTableByName
- IloCsvReader::getTable

Class IloTabuSearch

Definition file: ilsolver/iimmeta.h
Include file: <ilsolver/iimls.h>



This class implements a simple tabu search mechanism. The form of tabu search implemented is one where neighboring moves are forbidden depending on the states of two tabu lists that are maintained internally. These tabu lists maintain assignments to `IloNumVar` and `IloAnyVar` objects that have recently been removed from the solution ("out" assignments) and have recently been added to the solution ("in" assignments).

The tabu search mechanism uses the following rules:

- If a move proposes to add any assignments that were moved out of the solution less than or equal to `forbid` moves ago, the move is rejected.
- If a move proposes to remove any assignments that were moved in to the solution less than or equal to `keep` moves ago, the move is rejected.

These two rules are overridden if the current solution is of a lower cost than the best cost solution used to start the metaheuristic.

When a move is accepted and the `IloMetaHeuristic::notify` method is called for the tabu search metaheuristic, added new assignments and removed old assignments are recorded in the tabu lists so that they can be tested in the future.

A call to `IloMetaHeuristic::complete` "ages" the tabu lists by one iteration, so that from the point of view of the tabu restriction rules above, it appears that a move has been made. It returns `IloTrue` if the tabu list was empty before aging, and `IloFalse` otherwise.

See Also: IloMetaHeuristic, IloMetaHeuristicI

Constructor Summary	
`public`	`IloTabuSearch(IloEnv env, IloInt forbid, IloInt keep=0, IloNum step=1e-4)`

Method Summary	
`public IloNum`	`getAspirationStep() const`
`public IloInt`	`getForbidTenure() const`
`public IloInt`	`getKeepTenure() const`
`public void`	`setAspirationStep(IloNum step)`
`public void`	`setForbidTenure(IloInt t) const`
`public void`	`setKeepTenure(IloInt t) const`

Inherited Methods from `IloMetaHeuristic`
`complete, end, getDeltaCheck, getEnv, getImpl, getName, getObject, isFeasible, notify, operator=, reset, setName, setObject, start, test`

Constructors

`public` **`IloTabuSearch`**`(IloEnv env, IloInt forbid, IloInt keep=0, IloNum step=1e-4)`

This constructor creates a tabu search object associated with the environment `env`. The parameter `forbid` indicates the number of moves to maintain an old removed assignment on the tabu list. The parameter `keep`, if supplied, indicates the number of moves to maintain a new added assignment on the tabu list. The parameter `step` indicates how much better than the best solution found previously the current solution must be before the tabu status of any assignments is overridden.

Methods

`public IloNum` **`getAspirationStep`**`() const`

This member function returns the value of step specified in the constructor or through the previous call to `IloTabuSearch::setAspirationStep`.

`public IloInt` **`getForbidTenure`**`() const`

This member function returns the `forbid` value specified in the constructor or specified in the last call to `IloTabuSearch::setForbidTenure`.

`public IloInt` **`getKeepTenure`**`() const`

This member function returns the `keep` value specified in the constructor or specified in the last call to `IloTabuSearch::setKeepTenure`.

`public void` **`setAspirationStep`**`(IloNum step)`

This member function sets the value of the aspiration step to `step`.

`public void` **`setForbidTenure`**`(IloInt t) const`

This member function changes the number of moves that forbidden assignments remain on the tabu list. Any new assignments added remain on the list for `t` moves; the tenure of assignments already on the list is not changed.

`public void` **`setKeepTenure`**`(IloInt t) const`

This member function changes the number of moves that kept assignments remain on the tabu list. Any new assignments added remain on the list for `t` moves; the tenure of assignments already on the list is not changed.

Class IloTimer

Definition file: ilconcert/iloenv.h

Represents a timer.
An instance of `IloTimer` represents a timer in a Concert Technology model. It works like a stop watch. The timer report the CPU time. On multi threaded environment, we summed the CPU time used by each thread.

See Also: IloEnv

Constructor Summary	
`public`	`IloTimer(const IloEnv env)`

Method Summary	
`public IloEnv`	`getEnv() const`
`public IloNum`	`getTime() const`
`public void`	`reset()`
`public IloNum`	`restart()`
`public IloNum`	`start()`
`public IloNum`	`stop()`

Constructors

`public` **`IloTimer`**`(const IloEnv env)`

This constructor creates a timer.

Methods

`public IloEnv` **`getEnv`**`() const`

This constructor creates an instance of the class `IloTimer`

This member function returns the environment in which the invoking timer was constructed.

`public IloNum` **`getTime`**`() const`

This member function returns the accumulated time, in seconds, since one of these conditions:

- the first call of the member function `start` after construction of the invoking timer;
- the most recent call to the member function `restart`;
- a call to `reset`.

`public void` **`reset`**`()`

This member function sets the elapsed time of the invoking timer to 0.0. It also stops the clock.

`public IloNum` **`restart`**`()`

This member function returns the accumulated time, resets the invoking timer to 0.0, and starts the timer again. In other words, the member function `restart` is equivalent to the member function `reset` followed by `start`.

`public IloNum` **`start`**`()`

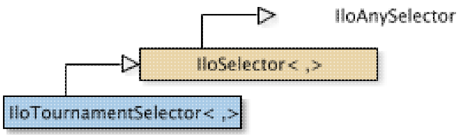
This member function makes the invoking timer resume accumulating time. It returns the time accumulated so far.

`public IloNum` **`stop`**`()`

This member function stops the invoking timer so that it no longer accumulates time.

Class IloTournamentSelector<,>

Definition file: ilsolver/iimmulti.h
Include file: <ilsolver/iim.h>



A selector which chooses objects following a tournament rule.
A *tournament selection* is one where objects compete to be selected and therefore a notion of pairwise comparable objects is needed. A tournament selector is defined by two objects: the *tournament size* (referred to as *t*) and a *comparator*. To select an object, the tournament selector selects *t* objects in a random unbiased way *with replacement*, meaning that the same object can be selected more than once. Then, the comparator is used to select the best of those *t* objects. In the case of ties (equal quality objects), the object selected first in the initial round will be chosen. The effect of the tournament size *t* is to prefer the "better" objects for higher *t*. *t=1* corresponds to random selection, but for normal usage, *t* is normally a small integer, as if it is too high, only the very best objects are ever selected.

When you create a tournament selector, you must pass a visitor object which can traverse the container from which you are selecting objects. However, in the case where a *default visitor* exists for the container in question, you can omit the visitor and the default will be used. In addition to the default visitors already in Solver, IIM provides default visitors for `IloSolutionPool` and `IloPoolProcArray`.

The following code shows how to declare a tournament selector and use it to build a pool processor:

```
IloTournamentSelector<IloSolution, IloSolutionPool> tsel(
  env, 2, IloBestSolutionComparator(env)
);
IloPoolProc selector = IloSelectSolutions(env, tsel, IloTrue);
```

In this example, better objective value solutions are preferred and the tournament size is two.

> **Note**
>
> An instance of `IloTournamentSelector` which has been transformed into a pool processor using `IloSelectSolutions` will always draw its random numbers from the random number generator of the solver on which it is executing.

See Also: IloSolutionPool, IloPoolProcArray, IloVisitor, IloComparator, IloRandomSelector, IloRouletteWheelSelector

Constructor Summary	
public	`IloTournamentSelector(IloEnv env, IloInt tournamentSize, IloComparator< IloObject > cmp, IloVisitor< IloObject, IloContainer > visitor=0)` Builds a tournament selector.

Method Summary	
`public IloComparator< IloObject >`	`getComparator() const` Delivers the comparator given at construction time.
`public IloInt`	`getTournamentSize() const`

	Delivers the tournament size.
public IloVisitor< IloObject, IloContainer >	getVisitor() const Delivers the visitor used by the selector.

Inherited Methods from `IloSelector`
select

Constructors

`public IloTournamentSelector(IloEnv env, IloInt tournamentSize, IloComparator< IloObject > cmp, IloVisitor< IloObject, IloContainer > visitor=0)`

Builds a tournament selector.

This constructor builds a selector from an environment `env` which will select objects according to a competitive "tournament" rule. This constructor uses the comparator `cmp` to compare objects. The parameter `tournamentSize` controls the number of randomly chosen objects to consider each time a selection is made. It is used to control the size of the competition and thus the bias of the selector towards preferred objects. An optional visitor `visitor` can be passed to traverse the container. If no visitor is specified, a default visitor will be used if it exists. If no default visitor exists, an exception of type `IloException` is raised.

Methods

`public IloComparator< IloObject > getComparator() const`

Delivers the comparator given at construction time.

This member function returns the comparator passed at construction time.

`public IloInt getTournamentSize() const`

Delivers the tournament size.

This member function returns the tournament size given at construction time.

`public IloVisitor< IloObject, IloContainer > getVisitor() const`

Delivers the visitor used by the selector.

This member function returns the visitor passed at construction time, or, if no visitor was passed, the default visitor for the container from which objects are being selected.

Class IloTranslator<,>

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

The `IloTranslator` template class is the base class for all translators. A translator is a class that implements a translation from an object of class `IloObjectIn` into an object of class `IloObjectOut` (using `operator()`). Translators allow you to easily define new predicates and evaluators on instances of class `IloObjectIn` given a predicate/evaluator on instances of class `IloObjectOut`. To define a new class of translator, use the macro `ILOTRANSLATOR`.

For more information, see Selectors.

See Also: operator<<

Method Summary	
`public IloObjectOut`	`operator()(const IloObjectIn & o, IloAny nu=0) const` Translates object of type `IloObjectIn` into an object of type `IloObjectOut`.

Methods

`public IloObjectOut` **`operator()`**`(const IloObjectIn & o, IloAny nu=0) const`

Translates object of type `IloObjectIn` into an object of type `IloObjectOut`.

This operator translates `o`, an object of type `IloObjectIn`, into an object of type `IloObjectOut`. The parameter `nu` allows you to add an optional context.

Class IloVisitor<,>

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

A visitor `IloVisitor<class IloObject,class IloContainer>` is a class that allows you to traverse each of the elements of type `IloObject` of a container class `IloContainer`. For example, a visitor can be used to specify how to traverse the set of variables (`IloIntVar`) of an array of variables (`IloIntVarArray`).

An instance of visitor can be initialized using the macros `ILOVISITOR0` and `ILODEFAULTVISITOR`.

For more information, see Selectors.

See Also: ILOVISITOR0, ILODEFAULTVISITOR, IloBestSelector

Class IloXmlContext

Definition file: ilconcert/iloxmlcontext.h

An instance of `IloXmlContext` allows you to serialize an `IloModel` or an `IloSolution` in XML.

You can write an `IloModel` using `IloXmlContext::writeModel`, write an `IloSolution` using `IloXmlContext::writeSolution`, or write both using `IloXmlContext::writeModelAndSolution`.

You can read an `IloModel` in XML using `IloXmlContext::readModel`, read an `IloSolution` in XML using `IloXmlContext::readSolution`, or read both using `IloXmlContext::readModelAndSolution`.

Other products should add their own serialization class and add them to the plug-in using the member functions `IloXmlContext::registerXML` and `IloXmlContext::registerXMLArray`.

Examples

For example, you can write:

```
IloModel model(env);
      IloSolution solution(env);
 ...;
IloXmlContext context(env);
context.writeModel(model, "model.xml");
context.writeSolution(solution, "solution.xml");
```

or you can write

```
IloModel model(env);
      IloSolution solution(env);
      IloXmlContext context(env);
      context.readModel(model, "model.xml");
      context.readSolution(solution, "solution.xml");
```

See Also: IloXmlReader, IloXmlWriter, IloXmlInfo

Constructor Summary	
`public`	`IloXmlContext(IloEnv env, const char * name=0)`
`public`	`IloXmlContext(IloXmlContextI * impl=0)`

Method Summary	
`public void`	`end()`
`public IloInt`	`getChildIdReadError() const`
`public const char *`	`getChildTagReadError() const`
`public IloIntArray`	`getIdListReadError() const`
`public IloXmlContextI *`	`getImpl() const`
`public IloInt`	`getParentIdReadError() const`
`public const char *`	`getParentTagReadError() const`
`public IloAnyArray`	`getTagListReadError() const`
`public const char *`	`getWriteError() const`

public int	getWritePrecision() const
public IloBool	readExtractable(IloXmlReader reader, IloXmlElement * element) const
public IloBool	readModel(IloModel model, istream & file) const
public IloBool	readModel(IloModel model, const char * fileName) const
public IloBool	readModelAndSolution(IloModel model, const char * modelFileName, IloSolution solution, const char * solutionFileName) const
public IloBool	readRtti(IloXmlReader reader, IloXmlElement * element) const
public IloBool	readSolution(IloSolution solution, istream & file) const
public IloBool	readSolution(IloSolution solution, const char * fileName) const
public IloBool	readSolutionValue(IloSolution solution, IloXmlElement * root, IloXmlReader reader) const
public void	registerXML(IloTypeIndex index, IloXmlInfo * xmlinfo) const
public void	registerXMLArray(IloXmlInfo * xmlinfo) const
public IloBool	setWriteMode(IloInt mode) const
public void	setWritePrecision(int writePrecision) const
public IloBool	writeExtractable(const IloExtractableI * it, IloXmlWriter writer, IloXmlElement * masterElement) const
public IloBool	writeModel(const IloModel model, const char * fileName) const
public IloBool	writeModelAndSolution(const IloModel model, const char * modelFileName, const IloSolution solution, const char * solutionFileName) const
public IloBool	writeRtti(const IloRtti * it, IloXmlWriter writer, IloXmlElement * masterElement) const
public IloBool	writeSolution(const IloSolution solution, const char * fileName) const
public void	writeSolutionValue(const IloExtractable it, const IloSolution solution, IloXmlWriter writer) const

Constructors

`public` **`IloXmlContext`**`(IloEnv env, const char * name=0)`

This constructor creates an XML context and makes it part of the environment `env`.

`public` **`IloXmlContext`**`(IloXmlContextI * impl=0)`

This constructor creates a XML context from its implementation object.

Methods

`public void end()`

This member function deletes the invoking XML context.

`public IloInt getChildIdReadError() const`

This member function returns the XML ID of the child unparsed XML element in cases where a problem occurs when reading an `IloModel`.

`public const char * getChildTagReadError() const`

This member function returns the XML tag of the child unparsed XML element in cases where a problem occurs when reading an `IloModel`

`public IloIntArray getIdListReadError() const`

This member function returns the XML ID list of the unparsed XML elements in cases where a problem occurs when reading an `IloModel`. The list is composed of the tags from the parent to the child elements.

`public IloXmlContextI * getImpl() const`

This member function returns the `IloXmlContextI` implementation.

`public IloInt getParentIdReadError() const`

This member function returns the XML ID of the parent unparsed XML element in cases where a problem occurs when reading an `IloModel`.

`public const char * getParentTagReadError() const`

This member function returns the XML tag of the parent unparsed XML element in cases where a problem occurs when reading an `IloModel`.

`public IloAnyArray getTagListReadError() const`

This member function returns the XML tag list of the unparsed XML elements in cases where a problem occurs when reading an `IloModel`. The list is composed of the tags from the parent to the child elements.

`public const char * getWriteError() const`

This member function returns the name of the extractable called in cases where a problem occurs when reading an `IloModel`.

`public int getWritePrecision() const`

This member function returns the write precision for floats

`public IloBool readExtractable(IloXmlReader reader, IloXmlElement * element) const`

xrefitem deprecated 7
See Also: IloXmlContext::readRtti

`public IloBool readModel(IloModel model, istream & file) const`

This member function reads `model` from an XML stream.

`public IloBool readModel(IloModel model, const char * fileName) const`

This member function reads `model` from the XML file `fileName`.

`public IloBool readModelAndSolution(IloModel model, const char * modelFileName, IloSolution solution, const char * solutionFileName) const`

This member function reads `model` and `solution` from their respective XML files, `modelFileName` and `solutionFileName`.

`public IloBool readRtti(IloXmlReader reader, IloXmlElement * element) const`

This member function tries to read all extractables from the XML element.

`public IloBool readSolution(IloSolution solution, istream & file) const`

This member function reads `solution` from an XML stream.

Note

This member function only works if a model has already been serialized.

`public IloBool readSolution(IloSolution solution, const char * fileName) const`

This member function reads `solution` from the XML file `fileName`.

Note

This member function only works if a model has already been serialized.

`public IloBool readSolutionValue(IloSolution solution, IloXmlElement * root,` IloXmlReader reader) const

This member function reads an `IloSolution` object from an XML element.

public void **registerXML**(IloTypeIndex index, IloXmlInfo * xmlinfo) const

This member function registers the serialization class of an extractable with a linked ID, usually its RTTI index. In write mode, the RTTI index is used to catch the correct serialization class.

In read mode, `IloXmlInfo::getTagName` is used to link the correct serialization class to the correct tag.

```
IlpXmlContext context(env);
       context.registerXML(IloAllDiffI::GetTypeIndex(), new (env) IloXmlInfo_AllDiff(context));
```

public void **registerXMLArray**(IloXmlInfo * xmlinfo) const

This member function registers the serialization class of an array of extractables with a linked ID.

```
context.registerXMLArray(new (env) IloXmlInfo_SOS2Array(context));
```

public IloBool **setWriteMode**(IloInt mode) const

This member function sets the write mode. The write mode can be set to `NoUnknown` or `EvenUnknown`. `NoUnknown` throws an exception if an attempt is made to serialize an unknown extractable. `EvenUnknown` writes a `Unknown` tag with the name of the extractable in a type attribute.

public void **setWritePrecision**(int writePrecision) const

This member function sets the write precision for floats. By default, there is no rounding mode on an `IloNum` or an `IloNumArray`. You can also choose the no rounding mode with the `IloNoRoundingMode` constant.

public IloBool **writeExtractable**(const IloExtractableI * it, IloXmlWriter writer, IloXmlElement * masterElement) const

xrefitem deprecated 8
See Also: IloXmlContext::writeRtti

public IloBool **writeModel**(const IloModel model, const char * fileName) const

This member function writes `model` to the file `fileName`in XML format.

public IloBool **writeModelAndSolution**(const IloModel model, const char * modelFileName, const IloSolution solution, const char * solutionFileName) const

This member function writes `model` to the file `modelFileName` and `solution` to the file `solutionFileName` in XML format.

```
public IloBool writeRtti(const IloRtti * it, IloXmlWriter writer, IloXmlElement *
masterElement) const
```

This member function writes a specified extractable. It is used from the serialization class of an extractable to write a embedded extractable.

The `IloOr` object calls this method on its constrained `vars`.

See Also: IloXmlInfo::writeRtti

```
public IloBool writeSolution(const IloSolution solution, const char * fileName)
const
```

This member function writes `solution` to the file `fileName` in XML format.

```
public void writeSolutionValue(const IloExtractable it, const IloSolution solution,
IloXmlWriter writer) const
```

This member function writes a specified extractable of a solution in XML. It is used from the serialization class of an extractable to write an embedded extractable.

See Also: IloXmlInfo::writeSolutionValue

Class IloXmlInfo

Definition file: ilconcert/iloxmlabstract.h

The class `IloXmlInfo` allows you to serialize an `IloModel` or an `IloSolution` in XML.

Constructor and Destructor Summary	
public	IloXmlInfo(IloXmlContextI * context, const char * version=0)
public	IloXmlInfo()

Method Summary	
public IloBool	checkAttExistence(IloXmlReader reader, IloXmlElement * element, const char * attribute)
public IloBool	checkExprExistence(IloXmlReader reader, IloXmlElement * element, const char * attribute, IloInt & id)
public IloXmlContextI *	getContext()
protected IloBool	getIntValArray(IloXmlReader reader, IloXmlElement * element, IloIntArray & intArray)
protected IloBool	getNumValArray(IloXmlReader reader, IloXmlElement * element, IloNumArray & numArray)
public IloBool	getRefInChild(IloXmlReader reader, IloXmlElement * element, IloInt & id)
public virtual const char *	getTag()
public virtual IloXmlElement *	getTagElement(IloXmlWriter writer, const IloRtti * exprI)
public static const char *	getTagName()
protected IloNumVar::Type	getVarType(IloXmlReader reader, IloXmlElement * element)
protected const char *	getVersion()
protected virtual IloRtti *	read(IloXmlReader reader, IloXmlElement * element)
public virtual IloExtractableArray *	readArrayFromXml(IloXmlReader reader, IloXmlElement * element)
public IloBool	readExtractable(IloXmlReader reader, IloXmlElement * element)
public virtual IloRtti *	readFrom(IloXmlReader reader, IloXmlElement * element)
public virtual IloExtractableI *	readFromXml(IloXmlReader reader, IloXmlElement * element)
public IloBool	readRtti(IloXmlReader reader, IloXmlElement * element)
public virtual IloBool	

	readSolution(IloXmlReader reader, IloSolution solution, IloXmlElement * element)
protected virtual IloExtractableI *	readXml(IloXmlReader reader, IloXmlElement * element)
protected virtual IloExtractableArray *	readXmlArray(IloXmlReader reader, IloXmlElement * element)
protected IloXmlElement *	setBoolArray(IloXmlWriter writer, const IloBoolArray Array)
public IloXmlElement *	setCommonArrayXml(IloXmlWriter writer, const IloExtractableArray * extractable)
public IloXmlElement *	setCommonValueXml(IloXmlWriter writer, const IloRtti * exprI)
public IloXmlElement *	setCommonXml(IloXmlWriter writer, const IloRtti * exprI)
protected IloXmlElement *	setIntArray(IloXmlWriter writer, const IloIntArray Array)
protected IloXmlElement *	setIntSet(IloXmlWriter writer, const IloIntSet Array)
protected IloXmlElement *	setNumArray(IloXmlWriter writer, const IloNumArray Array)
protected IloXmlElement *	setNumSet(IloXmlWriter writer, const IloNumSet Array)
protected void	setVersion(const char * version)
public void	setXml(IloXmlWriter writer, IloXmlElement * element, const IloRtti * exprI)
public virtual int	write(IloXmlWriter writer, const IloExtractableArray * extractable, IloXmlElement * masterElement)
public virtual IloBool	write(IloXmlWriter writer, const IloRtti * exprI, IloXmlElement * masterElement)
public IloBool	writeExtractable(IloXmlWriter writer, IloXmlElement * element, const IloExtractable extractable, const char * attribute=0)
public virtual IloBool	writeRef(IloXmlWriter writer, const IloRtti * exprI, IloXmlElement * masterElement)
public IloBool	writeRtti(IloXmlWriter writer, IloXmlElement * element, const IloRtti * rtti, const char * attribute=0)
public virtual void	writeSolution(IloXmlWriter writer, const IloSolution solution, const IloExtractable extractable)
public void	writeSolutionValue(IloXmlWriter writer, const IloSolution solution, IloXmlElement * element, const IloRtti * rtti, const char * attribute)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloSOS2Array array)

protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloSOS1Array array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloSemiContVarArray array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloConstraintArray array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloRangeArray array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloNumVarArray array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloIntSetVarArray array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloNumExprArray array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloIntExprArray array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloBoolVarArray array)
protected IloBool	writeVarArray(IloXmlWriter writer, IloXmlElement * element, IloIntVarArray array)
public virtual IloBool	writeXml(IloXmlWriter writer, const IloExtractableI * exprI, IloXmlElement * masterElement)
public virtual IloBool	writeXmlRef(IloXmlWriter writer, const IloExtractableI * exprI, IloXmlElement * masterElement)

Constructors and Destructors

public **IloXmlInfo**(IloXmlContextI * context, const char * version=0)

This constructor creates an instance of the handle class `IloXmlInfo` from a pointer to an instance of the undocumented implementation class `IloXmlContextI`.

public **IloXmlInfo**()

This constructor creates an empty instance of the handle class `IloXmlInfo`.

Methods

`public IloBool checkAttExistence(IloXmlReader reader, IloXmlElement * element, const char * attribute)`

Given a specified attribute, this member function checks `element` to establish whether the attribute exists. If the attribute does not exist, this member function throws an exception.

You can use this member function to dynamically validate an XML element.

`public IloBool checkExprExistence(IloXmlReader reader, IloXmlElement * element, const char * attribute, IloInt & id)`

Given a specified attribute, this member function checks `element` to establish whether the attribute exists, fills the `id`, and checks in the XML context memory whether an object with this `id` exists.

You can use this member function to dynamically validate an XML element.

Example: in the `read` method of the `IloDiff`, check that the `IdRef` object is already serialized

`public IloXmlContextI * getContext()`

This member function returns the related `IloXmlContextI` of the constructor.

`protected IloBool getIntValArray(IloXmlReader reader, IloXmlElement * element, IloIntArray & intArray)`

This member function returns the contained `IloIntArray` in the XML element `element`.

See Also: IloXmlReader::string2IntArray

`protected IloBool getNumValArray(IloXmlReader reader, IloXmlElement * element, IloNumArray & numArray)`

This member function returns the `IloNumArray` in the XML element `element`.

See Also: IloXmlReader::string2NumArray

`public IloBool getRefInChild(IloXmlReader reader, IloXmlElement * element, IloInt & id)`

Given an XML element, this member function checks for the first value `id` or `RefId` in the element and its children.

`public virtual const char * getTag()`

This member function returns the related XML tag.

`public virtual IloXmlElement * getTagElement(IloXmlWriter writer, const IloRtti * exprI)`

For backward compatibility with 2.0 and the XML for IloExtractable objects, if this method is not specialized, by default the getTagElement method with IloExtractableI will be called

`public static const char * getTagName()`

This static member function returns the linked XML tag of this serialization class.

`protected IloNumVar::Type getVarType(IloXmlReader reader, IloXmlElement * element)`

This member function returns the type of an `IloNumVar` - `IloFloat`, `IloInt`, or `IloBool` - in the XML element `element`.

`protected const char * getVersion()`

This member function returns the version of the object.

`protected virtual IloRtti * read(IloXmlReader reader, IloXmlElement * element)`

This member function reads an `IloRtti` from the given XML element.

This is the method to specialize for each serialization class

For backward compatibility with Concert 2.0 and the XML for IloExtractable objects, by default the method readXml with IloExtractableI will be called

`public virtual IloExtractableArray * readArrayFromXml(IloXmlReader reader, IloXmlElement * element)`

This member function reads an array of `IloRtti*` from the given XML element.

This is the method to specialize when writing a serialization class for an array of extractables.

`public IloBool readExtractable(IloXmlReader reader, IloXmlElement * element)`

xrefitem deprecated 4

`public virtual IloRtti * readFrom(IloXmlReader reader, IloXmlElement * element)`

This member function reads an `IloRtti` from the given XML element. It asks the XML context to read the extractable in the XML child element using a call to `IloXmlContext::readRtti`; it then calls `IloXmlInfo::readXml`.

For backward compatibility with Concert 2.0 and the XML for IloExtractable objects, by default the method readFromXml with IloExtractableI will be called `public virtual IloExtractableI * **readFromXml**(IloXmlReader reader, IloXmlElement * element)`

This member function reads an `IloRtti` from the given XML element. It asks the XML context to read the extractable in the XML child element using a call to `IloXmlContext::readRtti`; it then calls `IloXmlInfo::readXml`.

xrefitem deprecated 6

`public IloBool **readRtti**(IloXmlReader reader, IloXmlElement * element)`

This member function asks the XML context to read the `IloRtti` in the child element and then calls IloXmlInfo::readFromXml to read the parent extractable.

`public virtual IloBool **readSolution**(IloXmlReader reader, IloSolution solution, IloXmlElement * element)`

This member function reads a variable for `IloSolution` from the XML element `element`.

`protected virtual IloExtractableI * **readXml**(IloXmlReader reader, IloXmlElement * element)`

This member function reads an `IloRtti` from the given XML element.

This is the method to specialize for each serialization class

xrefitem deprecated 5

`protected virtual IloExtractableArray * **readXmlArray**(IloXmlReader reader, IloXmlElement * element)`

This member function reads an array of `IloRtti*` from the given XML element.

It is called by the XML context. It first asks the XML context to read from XML child elements using a call to `IloXmlContext::readRtti` and then calls `IloXmlInfo::readArrayFromXml`.

`protected IloXmlElement * **setBoolArray**(IloXmlWriter writer, const IloBoolArray Array)`

This member function creates an XML element containing the `IloBoolArray`.

See Also: IloXmlWriter::IntArray2String

`public IloXmlElement * **setCommonArrayXml**(IloXmlWriter writer, const IloExtractableArray * extractable)`

This member function creates a XML element with the common header for `IloExtractable` arrays.

`public IloXmlElement * **setCommonValueXml**(IloXmlWriter writer, const IloRtti * exprI)`

This member function creates an XML element with the given header for `IloRtti` from `IloSolution`.

```
public IloXmlElement * setCommonXml(IloXmlWriter writer, const IloRtti * exprI)
```

This member function creates an XML element with the common header for `IloRtti`.

```
protected IloXmlElement * setIntArray(IloXmlWriter writer, const IloIntArray Array)
```

This member function creates an XML element containing the `IloIntArray`.

See Also: IloXmlWriter::IntArray2String

```
protected IloXmlElement * setIntSet(IloXmlWriter writer, const IloIntSet Array)
```

This member function creates an XML element containing the `IloIntSet`.

See Also: IloXmlWriter::IntSet2String

```
protected IloXmlElement * setNumArray(IloXmlWriter writer, const IloNumArray Array)
```

This member function creates an XML element containing the `IloNumArray`.

See Also: IloXmlWriter::NumArray2String

```
protected IloXmlElement * setNumSet(IloXmlWriter writer, const IloNumSet Array)
```

This member function creates an XML element containing the `IloNumSet`.

See Also: IloXmlWriter::NumSet2String

```
protected void setVersion(const char * version)
```

This member function sets the version of the object.

```
public void setXml(IloXmlWriter writer, IloXmlElement * element, const IloRtti *
exprI)
```

This member function adds a name attribute and a ID attribute to the XML element.

```
public virtual int write(IloXmlWriter writer, const IloExtractableArray *
extractable, IloXmlElement * masterElement)
```

This member function writes the given `IloExtractableArray` in XML and adds it to the XML document of `writer`. This is the method to specialize when writing a serialization class

```
public virtual IloBool write(IloXmlWriter writer, const IloRtti * exprI,
IloXmlElement * masterElement)
```

This member function writes the `IloRtti` object `exprI` in XML and adds it to the XML document of the `IloXmlWriter` object `writer`.

For backward compatibility with Concert 2.0 and the XML for IloExtractable objects, by default the method writeXml with IloExtractableI will be called

```
public IloBool writeExtractable(IloXmlWriter writer, IloXmlElement * element, const
IloExtractable extractable, const char * attribute=0)
```

xrefitem deprecated 1

See `IloXmlContext::writeRtti(IloXmlWriter,IloXmlElement*,const IloRtti*,const char*)` instead. There is no longer need for the extractable argument.

```
public virtual IloBool writeRef(IloXmlWriter writer, const IloRtti * exprI,
IloXmlElement * masterElement)
```

This member function writes the `IloRtti` object `exprI` in XML as a reference.

For backward compatibility with Concert 2.0 and the XML for IloExtractable objects, by default the method writeXmlRef with IloExtractableI will be called

```
public IloBool writeRtti(IloXmlWriter writer, IloXmlElement * element, const
IloRtti * rtti, const char * attribute=0)
```

This member function writes an embedded extractable. Using the `getId()` method of the extractable, it adds an attribute with the ID in the XML element.

For example, used with `IloDiff`, this member function writes the expression and links it to the XML element via an `IdRef` attribute.

```
// using an IloDiffI* exprI:
    writeRtti(writer, element,
             (IloRtti*)exprI->getExpr1(),
             IloXmlAttributeDef::Expr1Id);
    writeRtti(writer, element,
             (IloRtti*)exprI->getExpr2(),
             IloXmlAttributeDef::Expr2Id);
```

See Also: IloXmlContext::writeRtti

```
public virtual void writeSolution(IloXmlWriter writer, const IloSolution solution,
const IloExtractable extractable)
```

This member function writes the specified extractable `extractable` from the `IloSolutionsolution` in XML format.

```
public void writeSolutionValue(IloXmlWriter writer, const IloSolution solution,
IloXmlElement * element, const IloRtti * rtti, const char * attribute)
```

This member function writes an embedded extractable of a solution in XML. Using the `getId()` method of the extractable, it adds an attribute with the ID in the XML element.

For example, used with `IloDiff`, this member function writes the expression and links it to the XML element via an `IdRef` attribute.

See Also: IloXmlContext::writeSolutionValue

```
protected IloBool writeVarArray(IloXmlWriter writer, IloXmlElement * element,
IloSOS2Array array)
```

This member function writes an `IloSOS2Array`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloSOS2s` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer, IloXmlElement * element,
IloSOS1Array array)
```

This member function writes an `IloSOS1Array`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloSOS1s` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer, IloXmlElement * element,
IloSemiContVarArray array)
```

This member function writes an `IloSemiContVarArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloSemiContVars` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer, IloXmlElement * element,
IloConstraintArray array)
```

This member function writes an `IloConstraintArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloConstraints` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer, IloXmlElement * element,
IloRangeArray array)
```

This member function writes an `IloRangeArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloRanges` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer, IloXmlElement * element,
IloNumVarArray array)
```

This member function writes an `IloNumVarArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloNumVars` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer, IloXmlElement * element,
IloIntSetVarArray array)
```

This member function writes an `IloIntSetVarArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloIntSetVars` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer, IloXmlElement * element,
IloNumExprArray array)
```

This member function writes an `IloNumExprArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloNumExprs` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer, IloXmlElement * element,
IloIntExprArray array)
```

This member function writes an `IloIntExprArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloIntExprs` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer, IloXmlElement * element,
IloBoolVarArray array)
```

This member function writes an `IloBoolVarArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloBoolVars` of `array`.

```
protected IloBool writeVarArray(IloXmlWriter writer, IloXmlElement * element,
IloIntVarArray array)
```

This member function writes an `IloIntVarArray`. It adds an attribute in the XML element `element` with the ID of `array`, serializes `array`, and, if necessary, serializes the `IloIntVars` of `array`.

Example using `IloSos` containing an `IloIntVarArray`:

```
 // Using an IloSOS1I* exprI;

          this.writeVarArray(writer,
                    element,
                    exprI->getVarArray(),
                    IloXmlAttributeDef::IdRef);
```

This sample adds an `IdRef` attribute on the SOS XML element, creates an XML element containing the `IloIntVarArray` with the list of `IloIntVar` IDs, and creates a list of XML elements for the `IloIntVars`.

```
public virtual IloBool writeXml(IloXmlWriter writer, const IloExtractableI * exprI,
IloXmlElement * masterElement)
```

This member function writes the `IloRtti` object `exprI` in XML and adds it to the XML document of the `IloXmlWriter` object `writer`.

xrefitem deprecated 2

```
public virtual IloBool writeXmlRef(IloXmlWriter writer, const IloExtractableI *
exprI, IloXmlElement * masterElement)
```

This member function writes the `IloRtti` object `exprI` in XML as a reference.

xrefitem deprecated 3

Class IloXmlReader

Definition file: ilconcert/iloreader.h

You can use an instance of `IloXmlReader` to read an `IloModel` or a `IloSolution` in XML format.

Constructor Summary	
public	IloXmlReader(IloEnv env, const char * fileName=0)
public	IloXmlReader(IloXmlReaderI * impl)

Method Summary	
public IloBool	checkRttiOfObjectById(IloTypeIndex RTTI, IloRtti * exprI)
public IloBool	checkRttiOfObjectById(IloTypeIndex RTTI, IloInt Xml_Id)
public IloBool	checkTypeOfObjectById(IloTypeInfo type, IloInt Xml_Id)
public IloBool	checkTypeOfObjectById(IloTypeInfo type, IloRtti * exprI)
public void	deleteAllocatedMemory(const char * pointer)
public void	deleteAllocatedMemory(char * pointer)
public IloXmlElement *	findElement(IloXmlElement * root, const char * tag, const char * attribute, const char * value)
public IloXmlElement *	findElementByTag(IloXmlElement * element, const char * tag)
public IloInt	getChildrenCardinal(IloXmlElement * element)
public IloEnv	getEnv()
public IloEnvI *	getEnvImpl()
public IloXmlElement *	getFirstSubElement(IloXmlElement * element)
public IloBool	getIntAttribute(IloXmlElement * element, const char * attribute, IloInt & value)
public IloBool	getNumAttribute(IloXmlElement * element, const char * attribute, IloNum & value)
public IloAny	getObjectById(IloInt id)
public IloXmlElement *	getRoot()
public IloIntArray *	getSerialized()
public IloIntArray *	getSolutionSerialized()
public IloBool	isSerialized(IloInt id)
public IloBool	openDocument()
public const char *	readAttribute(IloXmlElement * element, const char * attribute)
public const char *	readCData(IloXmlElement * element)
public const char *	readComment(IloXmlElement * element)
public const char *	readData(IloXmlElement * element)
public const char *	readText(IloXmlElement * element)
public void	setfileName(const char * fileName)

public IloInt	string2Int(const char * str)
public IloIntArray	string2IntArray(const char * str)
public IloIntRange	string2IntRange(IloXmlElement * element)
public IloIntSet	string2IntSet(const char * str)
public IloNum	string2Num(const char * str)
public IloNumArray	string2NumArray(const char * str)

Constructors

public **IloXmlReader**(IloEnv env, const char * fileName=0)

This constructor creates an `IloXmlReader` object and makes it part of the environment `env`.

The `fileName` is set to `0` by default.

public **IloXmlReader**(IloXmlReaderI * impl)

This constructor creates an XML reader from its implementation object.

Methods

public IloBool **checkRttiOfObjectById**(IloTypeIndex RTTI, IloRtti * exprI)

This method checks the RTTI of the given object.

public IloBool **checkRttiOfObjectById**(IloTypeIndex RTTI, IloInt Xml_Id)

This method checks the RTTI of the object referenced by the identifier `Xml_Id` in the XML. This object must already be serialized.

public IloBool **checkTypeOfObjectById**(IloTypeInfo type, IloInt Xml_Id)

This method checks the TypeInfo of the object referenced by the id in the XML. This object must have been already serialized.

public IloBool **checkTypeOfObjectById**(IloTypeInfo type, IloRtti * exprI)

This method checks the TypeInfo of the given object

public void **deleteAllocatedMemory**(const char * pointer)

This member function frees the memory that has been allocated by the XML reader using, for example, the IloXmlWriter::Int2String member function.

```
public void deleteAllocatedMemory(char * pointer)
```

This member function frees the memory that has been allocated by the XML reader using, for example, the IloXmlWriter::Int2String member function.

```
public IloXmlElement * findElement(IloXmlElement * root, const char * tag, const
char * attribute, const char * value)
```

This member function examines the XML element `root` to identify the XML child element denoted by `tag`, `attribute`, and `value`.

```
public IloXmlElement * findElementByTag(IloXmlElement * element, const char * tag)
```

This member function examines the XML element `element` to identify the XML child element denoted by `tag`.

```
public IloInt getChildrenCardinal(IloXmlElement * element)
```

This member function counts the number of child elements of the XML element `element`.

```
public IloEnv getEnv()
```

This member function gets the `IloEnv` of the object.

```
public IloEnvI * getEnvImpl()
```

This member function gets the implementation of the `IloEnv` of the object.

```
public IloXmlElement * getFirstSubElement(IloXmlElement * element)
```

This member function gets the first child in the XML element `element`.

```
public IloBool getIntAttribute(IloXmlElement * element, const char * attribute,
IloInt & value)
```

This member function checks the existence of `attribute` in the XML element `element` and converts it to an `IloInt`.

```
public IloBool getNumAttribute(IloXmlElement * element, const char * attribute,
IloNum & value)
```

This member function checks the existence of `attribute` in the XML element `element` and converts it to an `IloNum`.

`public IloAny` **`getObjectById`**`(IloInt id)`

This member function gets the already serialized object of the given identifier `id`.

```
IloDiff Diff(reader.getEnv(),
                 IloExpr((IloNumExprI*)reader.getObjectById(IdExpr1)),
                 IloExpr((IloNumExprI*)reader.getObjectById(IdExpr2)),
                 reader.readAttribute(element, IloXmlAttributeDef::Name));
```

The sample code creates a `IloDiff` from a XML element referencing its two expressions with the attributes `IdRef1` and `IdRef2`.

`public IloXmlElement *` **`getRoot`**`()`

This member function gets the XML root, that is, the XML document without the header.

`public IloIntArray *` **`getSerialized`**`()`

This member function gets the IDs of the serialized extractables and the unique IDs of the array of extractables that were serialized from the model.

`public IloIntArray *` **`getSolutionSerialized`**`()`

This member function gets the IDs of the serialized extractables and the unique IDs of the array of extractables that were serialized from the solution.

`public IloBool` **`isSerialized`**`(IloInt id)`

This member function checks whether the extractable with the ID `id` in the model has already been serialized.

`public IloBool` **`openDocument`**`()`

This member function opens the XML document specified in the constructor or with the `setFileName` method.

`public const char *` **`readAttribute`**`(IloXmlElement * element, const char * attribute)`

This member function returns the value of the `attribute` in the XML element `element`.

`public const char *` **`readCData`**`(IloXmlElement * element)`

This member function reads the CDATA of the XML element `element`.

`public const char *` **`readComment`**`(IloXmlElement * element)`

This member function returns the value of the comment in the XML element `element`.

`public const char * readData(IloXmlElement * element)`

This member function reads the data of the XML element `element`.

`public const char * readText(IloXmlElement * element)`

This member function returns the value of the text contained in the XML element `element`, independently of its origin (data or CDATA).

`public void setfileName(const char * fileName)`

This member function sets `fileName` as the file from which to read the XML.

`public IloInt string2Int(const char * str)`

This member function converts `str` into an `IloInt`.

`public IloIntArray string2IntArray(const char * str)`

This member function converts `str` into an `IloIntArray`.

`public IloIntRange string2IntRange(IloXmlElement * element)`

This member function converts `str` into an `IloIntRange`.

`public IloIntSet string2IntSet(const char * str)`

This member function converts `str` into an `IloIntSet`.

`public IloNum string2Num(const char * str)`

This member function converts `str` into an `IloNum`.

`public IloNumArray string2NumArray(const char * str)`

This member function converts `str` into an `IloNumArray`.

Class IloXmlWriter

Definition file: ilconcert/ilowriter.h

You can use an instance of `IloXmlWriter` to serialize an `IloModel` or an `IloSolution` in XML.

Constructor Summary	
public	IloXmlWriter(IloEnv env, const char * rootTag, const char * fileName=0)
public	IloXmlWriter(IloXmlWriterI * impl)

Method Summary	
public void	addAttribute(IloXmlElement * element, const char * attribute, const char * value)
public void	addCData(IloXmlElement * element, const char * CData)
public void	addComment(IloXmlElement * element, const char * comment)
public void	addElement(IloXmlElement * element)
public void	addSubElement(IloXmlElement * element, IloXmlElement * subElement)
public void	addText(IloXmlElement * element, const char * text)
public IloXmlElement *	createElement(const char * element)
public void	deleteAllocatedMemory(const char * pointer)
public void	deleteAllocatedMemory(char * pointer)
public IloEnv	getEnv()
public IloEnvI *	getEnvImpl()
public const char *	getfileName()
public IloXmlElement *	getRoot()
public IloIntArray *	getSerialized()
public IloIntArray *	getSolutionSerialized()
public const char *	Int2String(const IloInt number)
public const char *	IntArray2String(const IloIntArray intArray)
public const char *	IntSet2String(const IloIntSet intSet)
public IloBool	isSerialized(IloInt id)
public IloBool	isSolutionSerialized(IloInt id)
public const char *	Num2String(const IloNum number)
public const char *	NumArray2String(const IloNumArray numArray)
public const char *	NumSet2String(const IloNumSet numSet)
public void	setfileName(const char * fileName)
public IloInt	string2Int(const char * str)
public IloBool	writeDocument()

Constructors

```
public IloXmlWriter(IloEnv env, const char * rootTag, const char * fileName=0)
```

This constructor creates an `IloXmlWriter` object and makes it part of the environment `env`.

The `fileName` is set to `0` by default.

```
public IloXmlWriter(IloXmlWriterI * impl)
```

This constructor creates a XML writer object from its implementation object.

Methods

```
public void addAttribute(IloXmlElement * element, const char * attribute, const
char * value)
```

This member function adds an attribute of the specified value to the XML element.

```
public void addCData(IloXmlElement * element, const char * CData)
```

This member function adds a CDATA section to the XML element `element`.

```
public void addComment(IloXmlElement * element, const char * comment)
```

This member function adds `comment` to the XML element `element`.

```
public void addElement(IloXmlElement * element)
```

This member function adds the XML element `element` to the end of the XML.

```
public void addSubElement(IloXmlElement * element, IloXmlElement * subElement)
```

This member function adds a child element, `subElement`, to the XML element `element`.

```
public void addText(IloXmlElement * element, const char * text)
```

This member function adds `text` to the specified element.

```
public IloXmlElement * createElement(const char * element)
```

This member function creates an empty element with the given tag, `element`.

`public void **deleteAllocatedMemory**(const char * pointer)`

This member function frees the memory that has been allocated by the XML reader using, for example, the IloXmlWriter::Int2String member function.

`public void **deleteAllocatedMemory**(char * pointer)`

This member function frees the memory that has been allocated by the XML reader using, for example, the IloXmlWriter::Int2String member function.

`public IloEnv **getEnv**()`

This member function gets the `IloEnv` of the object.

`public IloEnvI * **getEnvImpl**()`

This member function gets the implementation of the `IloEnv` of the object.

`public const char * **getfileName**()`

This member function returns the name of the XML file

`public IloXmlElement * **getRoot**()`

This member function gets the root XML element of the XML document.

`public IloIntArray * **getSerialized**()`

This member function gets the IDs of the serialized objects of an `IloModel`.

`public IloIntArray * **getSolutionSerialized**()`

This member function gets the IDs of the serialized objects of an `IloSolution`.

`public const char * **Int2String**(const IloInt number)`

This member function converts the `IloInt` object `number` into a string, `const char*`.

`public const char * **IntArray2String**(const IloIntArray intArray)`

This member function converts the `IloIntArray` object `intArray` into a string, `const char*`.

`public const char * **IntSet2String**(const IloIntSet intSet)`

This member function converts the `IloIntSet` object `intSet` into a string, `const char*`.

`public IloBool **isSerialized**(IloInt id)`

This member function checks whether an object has been serialized.

`public IloBool **isSolutionSerialized**(IloInt id)`

This member function checks whether a solution object has already been serialized.

`public const char * **Num2String**(const IloNum number)`

This member function converts the `IloNum` object `number` into a string, `const char*`.

`public const char * **NumArray2String**(const IloNumArray numArray)`

This member function converts the `IloNumArray` object `numArray` into a string, `const char*`.

`public const char * **NumSet2String**(const IloNumSet numSet)`

This member function converts the `IloNumSet` object `numSet` into a string, `const char*`.

`public void **setfileName**(const char * fileName)`

This member function specifies `fileName` as the name of the XML file.

`public IloInt **string2Int**(const char * str)`

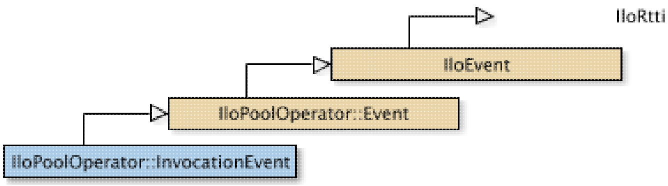
This member function converts `str` into an `IloInt`.

`public IloBool **writeDocument**()`

This member function outputs the XML to the file specified in the constructor or using the `setFileName` method. If null, this member function outputs on the `cout io`.

Class IloPoolOperator::InvocationEvent

Definition file: ilsolver/iimoperator.h
Include file: <ilsolver/iim.h>



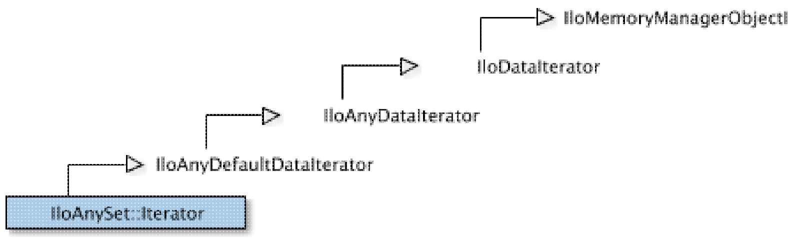
The event produced by pool operators when they are invoked.
When an operator is invoked, it produces an event of type `IloPoolOperator::InvocationEvent`. The event is emitted directly upon entering the operator. For example, assume that an operator was created using `op = op1 && op2`. When `op` is executed, first invocation events for `op` are produced, and then events for `op1` are produced. If `op1` then succeeds, an invocation event for `op2` will be produced when it begins execution.

See Also: operator&&, ILOIIMLISTENER0, IloListener

Inherited Methods from `Event`
`getOperator, getSolver`

Class IloAnySet::Iterator

Definition file: ilconcert/iloanyset.h



For IBM® ILOG® Solver: an iterator to traverse the elements of `IloAnySet`.
An instance of the nested class `IloAnySet::Iterator` is an iterator that traverses the elements of a finite set of pointers (an instance of `IloAnySet`).

See Also: IloAnySet

Constructor and Destructor Summary	
public	Iterator(const IloAnySetI * coll)
public	Iterator(const IloAnySet coll)
public	Iterator(IloGenAlloc * heap, const IloAnySetI * coll)
public	~Iterator()

Method Summary	
public IloBool	ok() const
public IloAny	operator*()
public void	operator++()

Constructors and Destructors

public **Iterator**(const IloAnySetI * coll)

.

public **Iterator**(const IloAnySet coll)

.

public **Iterator**(IloGenAlloc * heap, const IloAnySetI * coll)

.

public **~Iterator**()

.

Methods

`public IloBool **ok**() const`

This member function returns `IloTrue` if there is a current element and the invoking iterator points to it. Otherwise, it returns `IloFalse`.

To traverse the elements of a finite set of pointers, use the following code:

```
for(IloAnySet::Iterator iter(set); iter.ok(); ++iter){
        IloAny val = *iter;
        // do something with val
}
```

`public IloAny **operator***()`

This operator returns the current value.

`public void **operator++**()`

This operator advances the iterator to point to the next value in the dataset.

Class IloExplicitEvaluator::Iterator

Definition file: ilsolver/iimmulti.h
Include file: <ilsolver/iim.h>

An iterator which will iterate over all evaluated objects in an explicit evaluator.
This iterator iterates over all objects which have evaluations associated with them in an explicit evaluator. The objects are delivered in an arbitrary order, and do not necessarily correspond to the order in which the objects were given evaluations (via IloExplicitEvaluator::setEvaluation).

The iterator is robust to removals of objects (via IloExplicitEvaluator::removeEvaluation) at the iterator position, but if this is done, the iterator should not be incremented or an object will be skipped. Moreover, ok() should be used to determine if the iterator is still valid.

Constructor Summary	
public	Iterator(IloExplicitEvaluator< IloObject > ee) Creates an iterator to iterate over all evaluated objects in an explicit evaluator.

Method Summary	
public IloBool	ok() const Determines if the iteration is complete.
public IloObject	operator*() const Accesses the object at the current iterator position.
public Iterator &	operator++() Advances the iterator to the next object.

Constructors

public **Iterator**(IloExplicitEvaluator< IloObject > ee)

Creates an iterator to iterate over all evaluated objects in an explicit evaluator.

This constructor creates an iterator which will iterate (in no particular order) over all evaluated objects in the explicit evaluator ee.

Methods

public IloBool **ok**() const

Determines if the iteration is complete.

This member function should be called before each access to the iteration to determine if the current position is valid. When this function returns IloTrue, the iterator can safely be accessed (via operator *()), as long as the explicit evaluator is not changed before the access.

public IloObject **operator***() const

Accesses the object at the current iterator position.

This operator is used to deliver the current value of the iterator, which is an object which has an evaluation in the explicit evaluator specified in the constructor of the iterator.

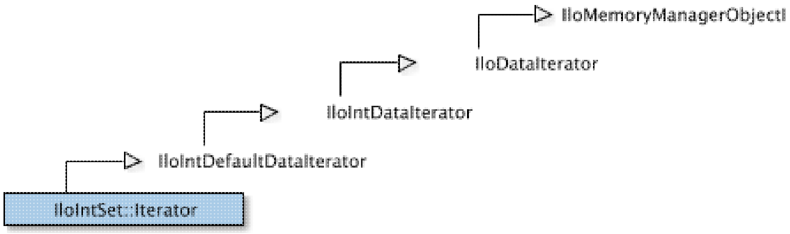
```
public Iterator & operator++()
```

Advances the iterator to the next object.

The operator moves the iterator to the next object which has an evaluation in the explicit evaluator being traversed by the iterator. Note that in the case where the object being pointed to by the iterator has just been deleted, this operator should not be called; the deletion process automatically advances the iterator in this case. A reference to the advanced iterator is returned.

Class IloIntSet::Iterator

Definition file: ilconcert/iloset.h



This class is an iterator that traverses the elements of a finite set of numeric values.
An instance of the nested class `IloIntSet::Iterator` is an iterator that traverses the elements of a finite set of numeric values (an instance of `IloIntSet`).

See Also: IloIntSet

Constructor and Destructor Summary	
public	Iterator(const IloIntSet coll)

Method Summary	
public IloBool	ok() const
public IloInt	operator*()
public void	operator++()

Constructors and Destructors

`public` **`Iterator`**`(const IloIntSet coll)`

Creates an iterator over the given set.

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloTrue` if there is a current element and the invoking iterator points to it. Otherwise, it returns `IloFalse`.

To traverse the elements of a finite set of pointers, use the following code:

```
for(IloIntSet::Iterator iter(set); iter.ok(); ++iter){
        IloInt val = *iter;
        // do something with val
}
```

`public IloInt` **`operator*`**`()`

This operator returns the current value.

```
public void operator++()
```

This operator advances the iterator to point to the next value in the dataset.

Class IloModel::Iterator

Definition file: ilconcert/ilomodel.h

Nested class of iterators to traverse the extractable objects in a model.
An instance of this nested class is an iterator capable of traversing the extractable objects in a model.

An iterator of this class differs from one created by the template `IloIterator`. Instances of `IloIterator` traverse all the extractable objects of a given class (specified by `E` in the template) within a given environment (an instance of `IloEnv`), whether or not those extractable objects have been explicitly added to a model. Instances of `IloModel::Iterator` traverse only those extractable objects that have explicitly been added to a given model (an instance of `IloModel`).

See Also: IloIterator, IloModel

Constructor Summary	
public	Iterator(const IloModel model)

Method Summary	
public IloBool	ok() const
public IloExtractable	operator*()
public void	operator++()

Constructors

`public` **`Iterator`**`(const IloModel model)`

This constructor creates an iterator to traverse the extractable objects in the model specified by `model`.

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloTrue` if there is a current element and the iterator points to it. Otherwise, it returns `IloFalse`.

`public IloExtractable` **`operator*`**`()`

This operator returns the current extractable object, the one to which the invoking iterator points.

`public void` **`operator++`**`()`

This operator advances the iterator to point to the next extractable object in the model.

Class IloSolution::Iterator

Definition file: ilconcert/ilosolution.h

It allows you to traverse the variables in a solution.
`Iterator` is a class nested in the class `IloSolution`. It allows you to traverse the variables in a solution. The iterator scans the objects in the same order as they were added to the solution.

This iterator is not robust. If the variable at the current position is deleted from the solution being iterated over, the behavior of this iterator afterward is undefined.

- `iter` can be safely used after the following code has executed:

```
IloExtractable elem = *iter;

++iter;

solution.remove(elem);
```

- `iter` cannot be safely used after the following code has executed:

```
solution.remove(*iter); // bad idea

++iter;
```

See Also: IloIterator, IloSolution

Constructor Summary	
public	Iterator(IloSolution solution)

Method Summary	
public IloBool	ok() const
public IloExtractable	operator*() const
public Iterator &	operator++()

Constructors

`public **Iterator**(IloSolution solution)`

This constructor creates an iterator to traverse the variables of `solution`. The iterator traverses variables in the same order they were added to `solution`.

Methods

`public IloBool **ok**() const`

This member function returns `IloTrue` if the current position of the iterator is a valid one. It returns `IloFalse` if all variables have been scanned by the iterator.

```
public IloExtractable operator*() const
```

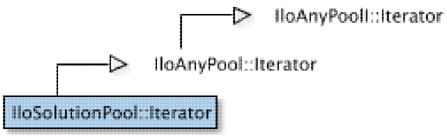
This operator returns the extractable object corresponding to the variable located at the current iterator position. If all variables have been scanned, this operator returns an empty handle.

```
public Iterator & operator++()
```

This operator moves the iterator to the next variable in the solution.

Class IloSolutionPool::Iterator

Definition file: ilsolver/iimpool.h
Include file: <ilsolver/iim.h>



brief Iterates over all the instances of `IloSolution` in the pool.

An instance of this class is an iterator capable of traversing all instances of `IloSolution` contained in the pool.

Constructor Summary	
`public`	`Iterator(IloSolutionPool pool)`

Method Summary	
`public IloBool`	`ok() const`
`public IloSolution`	`operator*() const`
`public Iterator &`	`operator++()`

Constructors

`public` **`Iterator`**`(IloSolutionPool pool)`

brief Creates an iterator which will iterate over a pool.

This constructor creates an iterator to traverse the pool `pool`.

Methods

`public IloBool` **`ok`**`() const`

brief Indicates whether all items have already been scanned.

This operator returns true if and only if items remain to be scanned.

`public IloSolution` **`operator*`**`() const`

brief Returns the current item.

This operator returns the current element, the `IloSolution` to which the invoking iterator points.

`public Iterator &` **`operator++`**`()`

brief Advances the iterator to the next pool element.

This operator advances the iterator to the next pool element and returns a reference to the invoking iterator.

Class IloExpr::LinearIterator

Definition file: ilconcert/iloexpression.h

An iterator over the linear part of an expression.
An instance of the nested class IloExpr::LinearIterator is an iterator that traverses the linear part of an expression.

Example

Start with an expression that contains both linear and non linear terms:

```
IloExpr e = 2*x + 3*y + cos(x);
```

Now define a linear iterator for the expression:

```
IloExpr::LinearIterator it = e.getLinearIterator();
```

That constructor creates a linear iterator initialized on the first linear term in `e`, that is, the term `(2*x)`. Consequently, a call to the member function `ok` returns `IloTrue`.

```
it.ok(); // returns IloTrue
```

A call to the member function `getCoef` returns the coefficient of the current linear term.

```
it.getCoef(); // returns 2 from the term (2*x)
```

Likewise, the member function `getVar` returns the handle of the variable of the current linear term.

```
it.getVar(); // returns handle of x from the term (2*x)
```

A call to the `operator++` at this point advances the iterator to the next linear term, `(3*y)`. The iterator ignores nonlinear terms in the expression.

```
++it; // goes to next linear term (3*y)
it.ok(); // returns IloTrue
it.getCoef(); // returns 3 from the term (3*y)
it.getVar(); // returns handle of y from the term (3*y)
++it; // goes to next linear term, if there is one in the expression
it.ok(); // returns IloFalse because there is no linear term
```

Method Summary	
public IloNum	getCoef() const
public IloNumVar	getVar() const
public IloBool	ok() const
public void	operator++()

Methods

public IloNum **getCoef**() const

This member function returns the coefficient of the current term.

`public IloNumVar getVar() const`

This member function returns the variable of the current term.

`public IloBool ok() const`

This member function returns `IloTrue` if there is a current element and the iterator points to it. Otherwise, it returns `IloFalse`.

`public void operator++()`

This operator advances the iterator to point to the next term of the linear part of the expression.

Class IloCsvReader::LineIterator

Definition file: ilconcert/ilocsvreader.h

Line-iterator for csv readers.
`LineIterator` is a nested class of the class `IloCsvReader`. It is to be used only with csv reader objects built to read a unique-table data file.

`IloCsvReader::LineIterator` allows you to step through all the lines of the csv data file (except blank lines and commented lines) on which the csv reader was created.

Constructor and Destructor Summary	
public	LineIterator()
public	LineIterator(IloCsvReader csv)

Method Summary	
public IloBool	ok() const
public IloCsvLine	operator*() const
public LineIterator &	operator++()

Constructors and Destructors

`public` **`LineIterator`**`()`

This constructor creates an empty `LineIterator` object. This object must be assigned before it can be used.

`public` **`LineIterator`**`(IloCsvReader csv)`

This constructor creates an iterator to traverse all the lines in the csv data file on which the csv reader `csv` was created.

The iterator does not traverse blank lines and commented lines.

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloTrue` if the current position of the iterator is a valid one.

It returns `IloFalse` if the iterator reaches the end of the table.

`public IloCsvLine` **`operator*`**`() const`

This operator returns the current instance of `IloCsvLine` (representing the current line in the csv file); the one to which the invoking iterator points.

```
public LineIterator & operator++()
```

This left-increment operator shifts the current position of the iterator to the next instance of `IloCsvLine` representing the next line in the file.

Class IloCsvTableReader::LineIterator

Definition file: ilconcert/ilocsvreader.h

Line-iterator for csv table readers.
`LineIterator` is a nested class of the class `IloCsvTableReader`. It allows you to step through all the lines of a table from a csv data file (except blank lines and commented lines) on which the table csv reader was created.

Constructor and Destructor Summary	
`public`	`LineIterator()`
`public`	`LineIterator(IloCsvTableReader csv)`

Method Summary	
`public IloBool`	`ok() const`
`public IloCsvLine`	`operator*() const`
`public LineIterator &`	`operator++()`

Constructors and Destructors

`public` **`LineIterator`**`()`

This constructor creates an empty `LineIterator` object.

This object must be assigned before it can be used.

`public` **`LineIterator`**`(IloCsvTableReader csv)`

This constructor creates an iterator to traverse all the lines in the table csv data file on which the csv reader `csv` was created.

The iterator does not traverse blank lines and commented lines.

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloTrue` if the current position of the iterator is a valid one.

It returns `IloFalse` if the iterator reaches the end of the table.

`public IloCsvLine` **`operator*`**`() const`

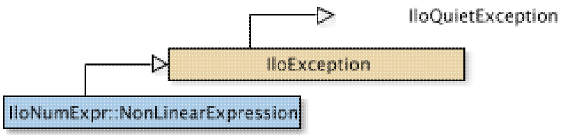
This operator returns the current instance of `IloCsvLine` (representing the current line in the csv file); the one to which the invoking iterator points.

`public LineIterator & operator++()`

This left-increment operator shifts the current position of the iterator to the next instance of `IloCsvLine` representing the next line in the file.

Class IloNumExpr::NonLinearExpression

Definition file: ilconcert/iloexpression.h



The class of exceptions thrown if a numeric constant of a nonlinear expression is set or queried.

Method Summary	
`public const IloNumExprArg`	`getExpression() const`

Inherited Methods from `IloException`
`end, getMessage`

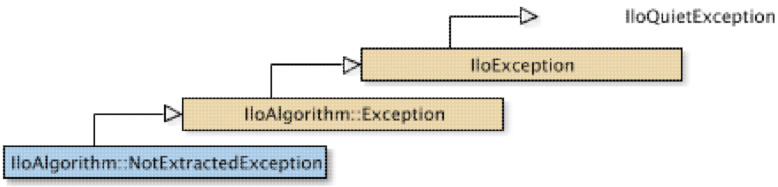
Methods

`public const IloNumExprArg` **`getExpression`**`() const`

The member function `getExpression` returns the expression involved in the exception.

Class IloAlgorithm::NotExtractedException

Definition file: ilconcert/iloalg.h



The class of exceptions thrown if an extractable object has no value in the current solution of an algorithm.
If an expression, numeric variable, objective, or array of extractable objects has no value in the current solution of an algorithm, this exception is thrown.

Constructor Summary	
public	`NotExtractedException(const IloAlgorithmI *, const IloExtractable)`

Method Summary	
`public const IloAlgorithmI *`	`getAlgorithm() const`
`public const IloExtractable &`	`getExtractable()`

Inherited Methods from `IloException`
`end, getMessage`

Constructors

`public` **`NotExtractedException`**`(const IloAlgorithmI *, const IloExtractable)`

The constructor `NotExtractedException` creates an exception thrown from the algorithm object `alg` for the extractable object `extr`.

Methods

`public const IloAlgorithmI *` **`getAlgorithm`**`() const`

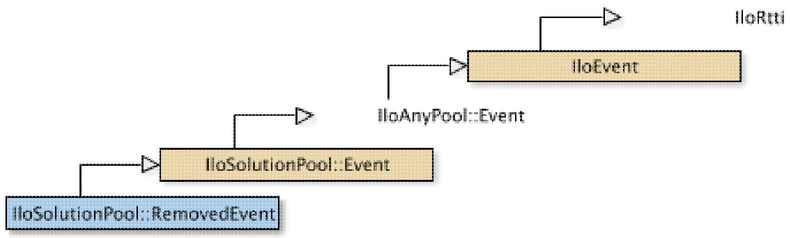
The member function `getAlgorithm` returns the algorithm from which the exception was thrown.

`public const IloExtractable &` **`getExtractable`**`()`

The member function `getExtractable` returns the extractable object that triggered the exception.

Class IloSolutionPool::RemovedEvent

Definition file: ilsolver/iimpool.h
Include file: <ilsolver/iim.h>



brief Event class used to notify the removal of an `IloSolution` from an `IloSolutionPool`.

This event class is used to notify any listeners that have been attached to the pool using `IloSolutionPool::addListener` whenever an object of type `IloSolution` is removed from an `IloSolutionPool` using `IloSolutionPool::remove`.

See Also: IloSolutionPool::addListener, IloSolutionPool::remove

Method Summary	
public IloSolution	getSolution() const

Inherited Methods from `Event`
getPool

Methods

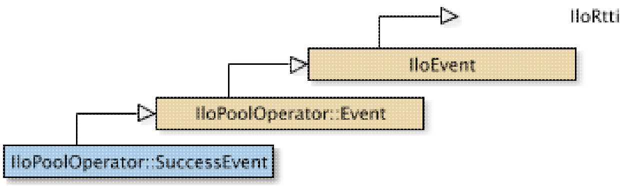
public IloSolution **getSolution**() const

brief Returns the removed `IloSolution`.

This function returns the removed object of type `IloSolution`.

Class IloPoolOperator::SuccessEvent

Definition file: ilsolver/iimoperator.h
Include file: <ilsolver/iim.h>



The event produced by pool operators when they are involved in the production of a solution.
When an operator succeeds, it produces an event of type `IloPoolOperator::SuccessEvent`. This event is produced when *all* operators involved in the creation of a solution have succeeded, just after the new solution has been stored.

For instance, if an operator `op` has been formed using `op = op1 && op2`, and when `op` is executed, `op1` succeeds and `op2` fails, this is treated as a global failure and no success events are issued. If both `op1` and `op2` have succeeded, however, success events would be produced for `op1`, `op2` and the combined operator `op`.

In the case where `op` has been formed using `op = op1 || op2` , if `op1` succeeds then success events are produced for operators `op1` and `op`. Otherwise, if `op2` succeeds, success events are produced for operators `op2` and `op`.

See Also: operator&&, operator||, ILOIIMLISTENER0, IloListener

Inherited Methods from `Event`
`getOperator, getSolver`

Class IloCsvReader::TableIterator

Definition file: ilconcert/ilocsvreader.h

Table-iterator of csv readers.
`TableIterator` is a nested class of the class `IloCsvReader`. It is to be used only for multitable files.

`IloCsvReader::TableIterator` allows you to step through all the tables of the multitable csv data file on which the csv reader was created.

Constructor and Destructor Summary	
`public`	`TableIterator(IloCsvReader csv)`

Method Summary	
`public IloBool`	`ok() const`
`public IloCsvTableReader`	`operator*() const`
`public TableIterator &`	`operator++()`

Constructors and Destructors

```
public TableIterator(IloCsvReader csv)
```

This constructor creates an iterator to traverse all the tables in the csv data file on which the csv reader `csv` was created.

Methods

```
public IloBool ok() const
```

This member function returns `IloTrue` if the current position of the iterator is a valid one.

It returns `IloFalse` if the iterator reaches the end of the table.

```
public IloCsvTableReader operator*() const
```

This operator returns the current instance of `IloCsvTable` (representing the current table in the csv file); the one to which the invoking iterator points.

```
public TableIterator & operator++()
```

This left-increment operator shifts the current position of the iterator to the next instance of `IloCsvTableReader` representing the next line in the file.

Enumeration IlcErrorType

Definition file: ilsolver/ilcerr.h
Include file: <ilsolver/ilosolver.h>

Solver associates an identifier with each error detected at execution time. Those identifiers are defined in the enumeration `IlcErrorType`.

As of version 5.0, Solver transforms an error into an instance of `IloSolver::SolverErrorException`. The member function `getErrorType` of that class will return the `IlcErrorType`.

See Also: IloSolver

Fields:

```
IlcError_arrayNullSize = 1

IlcError_collectionFilled

IlcError_nullCollectionSize

IlcError_extendExistingValue

IlcError_arrayBadSize

IlcError_arrayBadElement

IlcError_arrayIsNotOrdered

IlcError_badCollection

IlcError_emptyDomain

IlcError_undefValue

IlcError_nullObject

IlcError_alreadyUsed

IlcError_alreadyCalled

IlcError_notCalled

IlcError_mulBadOffset

IlcError_mulBoundVar

IlcError_mulNullValue

IlcError_failOutsideSolve

IlcError_noMoreChoicePoint

IlcError_failBadReturnValue

IlcError_usingUndefStrategy

IlcError_inSolve

IlcError_noMoreMemory

IlcError_deletingHeapObject

IlcError_internalError

IlcError_clauseInPropagation
```

IlcError_clauseNotInSolve

IlcError_unboundVarAfterMinimize

IlcError_badIndex

IlcError_badIndexInterval

IlcError_badIndexStep

IlcError_badArgument

IlcError_scheduleError

IlcError_scheduleAltError

IlcError_badPoints

IlcError_divideByZero

IlcError_keyAccess

IlcError_warnMemory

IlcError_emptyHandle

IlcError_noOpposite

IlcError_noMetaPost

IlcError_assignmentOfVariable

IlcError_notInProcess

IlcError_badEvent

IlcError_notAVariable

IlcError_divisionOfZero

IlcError_compatibility

IlcError_plannerError

IlcError_hybridError

IlcError_arrayRepeatedElement

IlcError_badSizeAllocator

IlcError_arrayAlreadyLocked

IlcError_differentManagers

IlcError_verticalError

IlcError_setAlreadyClosed

IlcError_setNotClosed

IlcError_predicateBadArity

IlcError_persistentHashTable

IlcError_overflowInExpression

IlcError_assertionFailed

IlcError_accessorFunctionBounds

IlcError_accessorPrototype

IlcError_notAllExtensibleAccessor

IlcError_getExtendedItemData

IlcError_incorrectSearchUse

IlcError_extensibleSet

IlcError_tooManyVar

IlcError_tooManyCt

IlcError_badConstraintLevel

IlcError_badFilterLevel

IlcError_changeFilterLevel

IlcError_continuousVar

IlcError_storeDuringSearch

Enumeration IlcFilterLevel

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>

The values in this enumeration control how much propagation is carried out.

Functions (such as `IlcAllDiff`, `IlcAllMinDistance`, `IlcAllNullIntersect`, `IlcDistribute`, `IlcPartition`, `IlcSequence`, for example) that return a constraint generally have two signatures: one without `IlcFilterLevel` as a parameter, one with `IlcFilterLevel` as an optional parameter. When you choose the signature without `IlcFilterLevel` as a parameter, then Solver uses the default filter level for that type of constraint to propagate the constraint returned by the function. The default filter level depends on the type of constraint. Types of constraints are defined by the enumeration `IlcFilterLevelConstraint`.

It is possible for you to change the default filter level globally by means of the member function `IloSolver::setDefaultFilterLevel`.

When you explicitly pass a filter level as a parameter to one of these functions, then you effectively mask the default filter level.

It is also possible to change the filter level yourself for a given constraint. You do so by means of the member function `IloSolver::setFilterLevel`.

In changing a filter level yourself, you can increase the filter level during a search, but you *cannot* decrease the filter level during a search. An attempt to decrease the filter level during a search will result in an error raising an exception

See Also: IlcAllDiff, IlcAllMinDistance, IlcAllNullIntersect, IlcDistribute, IlcEqUnion, IlcFilterLevelConstraint, IlcPartition, IlcSequence

Fields:

```
IlcLow = 0L

IloLowLevel = 0L

IlcBasic = 1L

IloBasicLevel = 1L

IlcMedium = 2L

IloMediumLevel = 2L

IlcExtended = 3L

IloExtendedLevel = 3L
```

Enumeration IlcFilterLevelConstraint

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>

The values in this enumeration recognize different types of constraints with respect to filter levels in propagation.

Functions (such as `IlcAllDiff`, `IlcAllMinDistance`, `IlcAllNullIntersect`, `IlcDistribute`, `IlcPartition`, `IlcSequence`, for example) that return a constraint generally have two signatures: one without `IlcFilterLevel` as a parameter, one with `IlcFilterLevel` as an optional parameter. When you choose the signature without `IlcFilterLevel` as a parameter, then Solver uses the default filter level for that type of constraint to propagate the constraint returned by the function. The default filter level depends on the type of constraint.

See Also: IlcAllDiff, IlcAllMinDistance, IlcAllNullIntersect, IlcBox, IlcDistribute, IlcPartition, IlcSequence

Fields:

`IlcAllDiffCt = 0L`

`IloAllDiffCt = 0L`

`IlcDistributeCt = 1L`

`IloDistributeCt = 1L`

`IlcSequenceCt = 2L`

`IloSequenceCt = 2L`

`IlcAllMinDistanceCt = 3L`

`IloAllMinDistanceCt = 3L`

`IlcPartitionCt = 4L`

`IloPartitionCt = 4L`

`IlcAllNullIntersectCt = 5L`

`IloAllNullIntersectCt = 5L`

`IlcEqUnionCt = 6L`

`IloEqUnionCt = 6L`

`IlcNotOverlapCt = 7L`

`IloNotOverlapCt = 7L`

`IlcBoxCt = 8L`

`IloBoxCt = 8L`

`IlcPairingCt = 9L`

`IloPairingCt = 9L`

Enumeration IlcFloatDisplay

Definition file: ilsolver/ilcerr.h
Include file: <ilsolver/ilosolver.h>

The values in this enumeration determine the format in which constrained floating-point variables are displayed. For example, the member function `IloSolver::setFloatDisplay` uses values from this enumeration to determine how to display constrained floating-point variables.

`IlcStandardDisplay` is the default value. When a constrained floating-point variable is displayed in this format, its minimal and maximal values are rounded to the nearest value, and the variable is displayed as an interval defined by these values, like this: `[min max]`.

`IlcIntScientific` displays a constrained floating-point variable as an interval defined by its minimum and maximum values, like this: `[min max]`. The minimal value is rounded toward negative infinity (-?); the maximal value is rounded toward positive infinity (+?). The values `min` and `max` are displayed in scientific notation `d.ddde+dd` where `d` represents a digit and `e` the base for exponentiation.

`IlcIntFixed` displays a constrained floating-point variable as an interval defined by its minimum and maximum values, like this: `[min max]`. The minimal value is rounded toward negative infinity (-?); the maximal value is rounded toward positive infinity (+?). The values `min` and `max` are displayed in fixed-precision notation `ddddd.dd` where `d` represents a digit.

`IlcBasScientific` displays a constrained floating-point variable as an interval defined by a base and two other values: `[base + [delta1 delta2]]` representing the interval `[(base + delta1) (base + delta2)]` . Two cases may arise: the interval contains an integer; the interval does not contain an integer.

- The interval contains an integer. In this case, the `base` is this integer, `delta1` is negative, and `delta2` is positive.
- The interval does not contain an integer. In this case, the `base` represents the common part. For example, in `IlcBasScientific`, a constrained floating-point variable between `1.23456789` and `1.23456890` will be displayed like this: `[1.23456 + [0.78899999998900e-5..0.88999999999700e-5]]`

If the minimal and maximal values are too far apart and consequently to display a base makes no sense, then a constrained floating-point variable in `IlcBasScientific` format is displayed like this: `[min max]`.

`IlcBasFixed` displays a constrained floating-point variable as an interval defined by a base and two other values: `[base + [delta1 delta2]]`. Two cases may arise: the interval contains an integer; the interval does not contain an integer.

- The interval contains an integer. In this case, the `base` is this integer, `delta1` is negative, and `delta2` is positive. For example, in `IlcBasFixed`, a constrained floating-point variable between `0.99` and `1.01` will be displayed like this: `[1.0 + [-0.100000000000000089e-1..+0.10000000000000009e-1]]`
- The interval does not contain an integer. In this case, the `base` represents the common part.

If the minimal and maximal values are too far apart and consequently to display a base makes no sense, then a constrained floating-point variable in `IlcBasFixed` format is displayed like this: `[min max]`.

See Also: IloSolver

Fields:

`IlcStandardDisplay = 0`

`IlcIntScientific = 1`

`IlcIntFixed = 2`

`IlcBasScientific = 3`

```
IlcBasFixed = 4
```

Enumeration IlcPruneMode

Definition file: ilsolver/search.h
Include file: <ilsolver/ilosolver.h>

The values in this enumeration indicate why Solver prunes the nodes of a search tree.

See Also: IlcSearchMonitorI

Fields:

`searchNotFailed`

`searchFailedNormally`

`killedBySelector`

`killedByLimit`

`killedByLabel`

`killedByEvaluator`

`killedByExit`

Enumeration Status

Definition file: ilconcert/iloalg.h

An enumeration for the class `IloAlgorithm`.
`IloAlgorithm` is the base class of algorithms in Concert Technology, and `IloAlgorithm::Status` is an enumeration limited in scope to the class `IloAlgorithm`. The member function `IloAlgorithm::getStatus` returns a status showing information about the current model and the solution.

`Unknown` specifies that the algorithm has no information about the solution of the model.

`Feasible` specifies that the algorithm found a feasible solution (that is, an assignment of values to variables that satisfies the constraints of the model, though it may not necessarily be optimal). The member functions `IloAlgorithm::getValue` access this feasible solution.

`Optimal` specifies that the algorithm found an optimal solution (that is, an assignment of values to variables that satisfies all the constraints of the model and that is proved optimal with respect to the objective of the model). The member functions `IloAlgorithm::getValue` access this optimal solution.

`Infeasible` specifies that the algorithm proved the model infeasible; that is, it is not possible to find an assignment of values to variables satisfying all the constraints in the model.

`Unbounded` specifies that the algorithm proved the model unbounded.

`InfeasibleOrUnbounded` specifies that the model is infeasible or unbounded.

`Error` specifies that an error occurred and, on platforms that support exceptions, that an exception has been thrown.

See Also: IloAlgorithm, operator<<

Fields:

`Unknown`

`Feasible`

`Optimal`

`Infeasible`

`Unbounded`

`InfeasibleOrUnbounded`

`Error`

Enumeration IloDeleterMode

Definition file: ilconcert/iloenv.h

An enumeration to set the mode of an `IloDeleter`.
This enumeration allows you to set the `IloDeleter` mode. The modes `IloRecursiveDeleterMode` and `IloSmartDeleterMode` are not documented and should not be used.

You can set the mode using the member function IloEnv::setDeleter. For a description of deletion in IBM® ILOG® Concert Technology, refer to Deletion of Extractables.

Fields:

```
IloLinearDeleterMode = 0

IloSafeDeleterMode = 1

IloRecursiveDeleterMode = 2

IloSmartDeleterMode = 2
```

Enumeration Type

Definition file: ilconcert/iloexpression.h

An enumeration for the class IloNumVar.
This nested enumeration enables you to specify whether an instance of `IloNumVar` is of type integer (`Int`), Boolean (`Bool`), or floating-point (`Float`).

Programming Hint

For each enumerated value in `IloNumVar::Type`, there is a predefined equivalent C++ `#define` in the Concert Technology include files to make programming easier. For example, instead of writing `IloNumVar::Int` in your application, you can write `ILOINT`. Likewise, `ILOFLOAT` is defined for `IloNumVar::Float` and `ILOBOOL` for `IloNumVar::Bool`.

See Also: IloNumVar

Fields:

```
Int = 1

Float = 2

Bool = 3
```

Enumeration Sense

Definition file: ilconcert/ilolinear.h

Specifies objective as minimization or maximization.
An instance of the class `IloObjective` represents an objective in a model. This nested enumeration is limited in scope to that class, and its values specify the sense of an objective; that is, whether it is a minimization (`Minimize`) or a maximization (`Maximize`).

See Also: IloObjective

Fields:

```
Minimize = 1

Maximize = -1
```

Enumeration FailureStatus

Definition file: ilsolver/ilosolverhandle.h
Include file: <ilsolver/ilosolver.h>

The values in this enumeration indicate the failure status of an instance of `IloSolver` after an unsuccessful IloSolver::solve or IloSolver::next. The member function IloSolver::getFailureStatus is used to query the failure status of an instance of `IloSolver`.

See Also: IloSolver

Fields:

`searchHasNotFailed`

`searchFailedNormally`

`searchStoppedByLimit`

`searchStoppedByLabel`

`searchStoppedByExit`

`unknownFailureStatus`

Enumeration IloSynchronizeMode

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>

The values in this enumeration indicate how Solver should synchronize changes in a model with its search for solutions. In general, you cannot modify the current extracted model during a search. In other words, you cannot change a model while the search is *active*. A search is active during the call of `IloSolver::solve` and between a call of `IloSolver::startNewSearch` and the corresponding call of `IloSolver::endSearch`. You can change the model only at top level (that is, not nested).

Solver buffers changes you make in a model; the buffered changes are *not* applied immediately. Certain member functions check that buffer:

- The member function `IloSolver::solve`checks the buffer at the beginning of its search.
- The member function `IloSolver::startNewSearch`checks the buffer before it begins a new search.

The values of the enumeration `IloSynchronize` mode serve as parameters to such member functions as `IloSolver::startNewSearch`.

- `IloSynchronizeAndRestart` when an instance of `IloSolver` checks the change buffer, makes the solver discard the current model, discard information about the current search, possibly extract the new model including these changes (if necessary), and restart its search “from scratch.”
- `IloSynchronizeAndContinue` indicates that the member function `IloSolver::startNewSearch` should check the buffer for changes in the model before it begins a new search. For example, in this mode, the member function will check for such changes in the model as adding or removing an objective (an instance of `IloObjective`) or a constraint (an instance of `IloConstraint`). Then the solver should synchronize those changes with its search results if possible. When this mode is in effect, if it is not possible to take the changes into account without extracting the model again, Solver will throw an exception, `IloSolver::IloSynchronizeAndContinueNotPossible` .

Example

The following lines highlight changes in the model, buffering, and Solver search.

```
IloEnv env;
IloModel model(env);

// add extractable objects to model

IloSolver solver(model);
solver.solve();

// modify the model (A)

solver.startNewSearch(mygoal, mode);
solver.next();
solver.endSearch();
```

Solver buffers the changes in part (A). The buffered changes will be taken into account only at top level (not nested).

While part (A) is going on, (that is, while you are making changes to the model), the solver (the instance of `IloSolver`) is not notified about those changes. Instead, Solver silently buffers the changes that will be checked.

See Also: IloSolver

Fields:

`IloNoSynchronization`

`IloSynchronizeAndRestart`

`IloSynchronizeAndContinue`

Global function IloPiecewiseLinear

```
public IloNumExprArg IloPiecewiseLinear(const IloNumExprArg node, const IloNumArray
point, const IloNumArray slope, IloNum a, IloNum fa)
public IloNumExprArg IloPiecewiseLinear(const IloNumExprArg node, IloNum
firstSlope, const IloNumArray point, const IloNumArray value, IloNum lastSlope)
```

Definition file: ilconcert/iloexpression.h

Represents a continuous or discontinuous piecewise linear function.
The function `IloPiecewiseLinear` creates an expression node to represent a continuous or discontinuous piecewise linear function *f* of the variable *x*. The signatures of this overloaded function support two different ways to specify piecewise linear functions. One approach specifies the breakpoints and slopes of the segments of the PWL function. The other approach specifies the breakpoints and function values of the segments. Both approaches can specify either continuous or discontinuous piecewise linear functions.

However, the user must take care with the approach that uses breakpoints and slopes to specify a discontinuous piecewise linear function **uniquely**. For further explanation about unique specification of discontinuous piecewise linear functions, see the topic *Discontinuous piecewise linear functions* in the *IBM ILOG CPLEX User's Manual*.

In the approach using breakpoints and slopes to specify a PWL function, the array `point` contains the *n* breakpoints of the function such that *point[i-1] <= point[i] for i = 1, . . . , n-1*. The array `slope` contains the *n+1* slopes of the *n+1* segments of the function. The values `a` and `fa` must be coordinates of a point such that `fa = f(a)`.

The argument `a` cannot be the coordinate of a step. In other words, `a` must **not** be the `point[i]` if `point[i] == point[i-1]` or if `point[i] == point[i+1]`.

Example

```
IloPiecewiseLinear(x, IloNumArray(env, 2, 10., 20.),
                   IloNumArray(env, 3, 0.3, 1., 2.),
                   0, 0);
```

That expression defines a piecewise linear function *f* having two breakpoints at *x = 10* and *x = 20*, and three segments; their slopes are *0.3, 1,* and *2*. The first segment has infinite length and ends at the point *(x = 10, f(x) = 3)* since *f(0) = 0*. The second segment starts at the point *(x = 10, f(x) = 3)* and ends at the point *(x = 20, f(x) = 13)*, where the third segment starts.

For the approach that defines a piecewise linear function by breakpoints and values, the array `point` also contains the *n* breakpoints of the PWL function. The array `value` contains the corresponding *n* values of the function. The argument `firstSlope` specifies the slope of the segment to the left of the first breakpoint; the argument `lastSlope` specifies the sleop of the segment to the right of the final breakpoint.

For an example and further explanation of this approach of specification by breakpoint and value, see the topic *Discontinuous piecewise linear functions* in the *IBM ILOG CPLEX User's Manual*.

Global function IlcChooseMinMinInt

`public IlcInt IlcChooseMinMinInt(const IlcIntVarArray vars)`

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the smallest minimum from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseIntIndex, IloGenerate

Global function IlcMax

```
public IlcFloatExp IlcMax(const IlcIntSetVar aSet, const IlcIntToFloatExpFunction
f, const IlcBool makeEmptySetPropagation=IlcTrue)
```

Definition file: ilsolver/setcst.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a floating-point expression constrained to be the maximum returned by the function `f` over its domain, `aSet`.

See Also: IlcIntSetVar

Global function IlcMax

```
public IlcIntExp IlcMax(const IlcIntSetVar aSet, const IlcIntToIntFunction F, const
IlcBool makeEmptySetPropagation=IlcTrue)
public IlcIntExp IlcMax(const IlcIntSetVar aSet, const IlcIntToIntExpFunction F,
const IlcBool makeEmptySetPropagation=IlcTrue)
public IlcIntExp IlcMax(const IlcAnySetVar aSet, const IlcAnyToIntFunction F, const
IlcBool makeEmptySetPropagation=IlcTrue)
public IlcIntExp IlcMax(const IlcAnySetVar aSet, const IlcAnyToIntExpFunction F,
const IlcBool makeEmptySetPropagation=IlcTrue)
```

Definition file: ilsolver/setcst.h
Include file: <ilsolver/ilosolver.h>

These functions create and return a new constrained expression equal to the greatest of the values returned by the function `F` applied to the elements assigned to the constrained set variable `setVar`.

$$y = \max_{x \in \mathrm{setVar}} F(x)$$

The value returned by the function `F` can be an integer value (`IlcInt`) or an integer constrained expression (`IlcIntExp`).

The minimum value of the expression is the greatest value returned by `F` when applied to the required elements of the variable `setVar`. If there is no required element, the minimum value of the expression corresponds to the least value returned by `F` when applied to the possible elements of `setVar`. When `F` returns an integer expression, the maximum value of the expression is computed with the lower bounds of `F(x)`.

The maximum value of the expression is the greatest value returned by `F` when applied to the possible elements of the variable `setVar`. If `F` returns an integer expression, it corresponds to the greatest upper bound of `F(x)`.

Because the maximum value of an empty set has no meaning, the bounds of this expression are computed only when the set variable is surely not empty, that is, when its cardinality is greater than 0 (zero). The initial bounds of the expression are the minimum and the maximum value returned by `F` when applied to the possible elements of `setVar`.

See Also: IlcAnySetVar, IlcIntExp, IlcIntSetVar

Global function IlcMax

```
public IlcIntExp IlcMax(IlcInt exp1, const IlcIntExp exp2)
public IlcFloat IlcMax(const IlcFloat exp1, const IlcFloat exp2)
public IlcIntExp IlcMax(const IlcIntExp exp1, IlcInt exp2)
public IlcIntExp IlcMax(const IlcIntExp exp1, const IlcIntExp exp2)
public IlcIntExp IlcMax(const IlcIntVarArray array)
public IlcInt IlcMax(const IlcIntArray array)
public IlcInt IlcMax(const IlcInt exp1, const IlcInt exp2)
public IlcFloatExp IlcMax(const IlcFloatExp exp1, IlcFloat exp2)
public IlcFloatExp IlcMax(IlcFloat exp1, const IlcFloatExp exp2)
public IlcFloatExp IlcMax(const IlcFloatExp exp1, const IlcFloatExp exp2)
public IlcFloatExp IlcMax(const IlcFloatVarArray array)
public IlcFloat IlcMax(const IlcFloatArray array)
public IlcIntToIntStepFunction IlcMax(const IlcIntToIntStepFunction & f1, const
IlcIntToIntStepFunction & f2)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function returns the maximum of its argument or arguments.

When its argument is an array of constrained expressions, it creates a new constrained expression equal to the maximum of the elements in that array.

When its arguments include at least one constrained expression, it creates a new constrained expression equal to the maximum of the arguments.

When both its arguments are numeric (that is, values of type `IloInt` or `IloNum`), it simply creates and returns a value of that type.

See Also: IlcFloatExp, IlcFloatVarArray, IlcIntExp, IlcIntToIntStepFunction, IlcIntVarArray, IlcMin

Global function IlcMax

`public IlcIntExp IlcMax(const IlcIntSetVar aSet, const IlcBool makeEmptySetPropagation=IlcTrue)`

Definition file: ilsolver/setcst.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a new constrained expression equal to the greatest of the integer elements that are assigned to the constrained set variable `setVar`.

$$y = \max_{x \in \text{setVar}} x$$

The minimum value of the expression is the largest required integer element of the variable `setVar`, or, if there is none, the least possible element.

The maximum value of the expression is the largest possible integer element of the variable `setVar`.

Because the maximum value of an empty set has no meaning, the bounds of this expression are computed only when the set variable is surely not empty, that is, when its cardinality is greater than 0 (zero). The initial bounds of the expression are the minimum and the maximum possible elements.

See Also: IlcIntExp, IlcIntSetVar

Global function IloFloor

```
public IloEvaluator< IloObject > IloFloor(IloEvaluator< IloObject > eval)
```

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

This function creates a composite `IloEvaluator<IloObject>` instance. This evaluator returns the largest integer value not greater than the float value returned by the evaluator given as argument.

For more information, see Selectors.

Global function IloFloor

```
public IloNum IloFloor(IloNum val)
```

Definition file: ilconcert/iloenv.h

Returns the largest integer value not greater than the argument.
This function computes the largest integer value not greater than `val`.

Examples:

```
IloFloor(IloInfinity) is IloInfinity.
IloFloor(-IloInfinity) is -IloInfinity.
IloFloor(0) is 0.
IloFloor(0.4) is 0.
IloFloor(-0.4) is -1.
IloFloor(0.5) is 0.
IloFloor(-0.5) is -1.
IloFloor(0.6) is 0.
IloFloor(-0.6) is -1.
```

Global function IloBFSEvaluator

```
public IloNodeEvaluator IloBFSEvaluator(const IloEnv env, const IloNumVar var,
IloNum step=0)
```

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a *node evaluator* that implements a best first search in a Concert Technology model.

Nodes are evaluated according to the variable `var`. As long as the minimum of `var` is no greater than the minimum of the evaluation of each open nodes `+ step`, the search continues as a depth-first search. If the opposite is true, the goal manager postpones the evaluation of the current node and jumps to the best open node.

This function returns an instance of `IloNodeEvaluator` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the node evaluator that it returns as an instance of `IlcNodeEvaluator` for use during a Solver search.

For more information, see `IloNullIntersect`.

See Also: IlcNodeEvaluator, IloNodeEvaluator, IloNullIntersect

Global function IlcChooseFirstUnboundAnySet

```
public IlcInt IlcChooseFirstUnboundAnySet(const IlcAnySetVarArray vars)
```

Definition file: ilsolver/ilcset.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the first unbound constrained variable that it encounters in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseAnySetIndex, IloGenerate

Global function IloSwap

```
public IloNHood IloSwap(IloEnv env, IloNumVarArray vars, const char * name=0)
```

Definition file: ilsolver/iimnhood.h
Include file: <ilsolver/iimls.h>

This function returns a neighborhood that can be used to swap variable values in a local search problem.

The function defines a “swapping” neighborhood over the variables specified in `vars`. Specifically, for each pair of indices `a`, `b` drawn from `[0, vars.getSize())`, such that `a < b`, a swap of the values of `vars[a]` and `vars[b]` is present in the neighborhood.The optional argument `name`, if supplied, becomes the name of the returned neighborhood.

See Also: IloNHood, IloNHoodI

Global function IlcImpactInformation

```
public IlcConstraintAggregator IlcImpactInformation(IloSolver solver,
IloIntVarArray branchVarArray, IloIntVarArray observedVarArray=0)
```

Definition file: ilsolver/ilosolverhandle.h
Include file: <ilsolver/ilosolver.h>

This aggregator maintains impact information on the variables in array `x` when they are instantiated with the following goals:

```
IlcGoal IlcSetValue(IlcIntvar x, IlcInt v);
IlcGoal IlcRemoveValue(IlcIntVar x, IlcInt v);
```

For more information on impacts, see *"Using Impacts during Search"* in the *IBM ILOG Solver User's Manual*.

See Also: IlcSetValue, IlcRemoveValue

Global function IloOrGoal

```
public IloGoal IloOrGoal(const IloEnv env, const IloGoal g1, const IloGoal g2,
IloAny label=0)
```

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal. This goal represents the disjunction (that is, logical OR) between its parameters, `g1` and `g2`. The optional argument `label` is a name for the goal (that is, a label for the choice point). If you do not use a `label`, you can replace the code `IloOrGoal(env, g1, g2)` by `g1 || g2`.

If you would like to represent the disjunction of two constraints (rather than two goals), then you should consider an instance of the class `IloOr`.

When it takes an instance of the class `IloEnv` as a parameter, it returns an instance of `IloGoal` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the goal that it returns as an instance of `IlcGoal` for use during a Solver search.

See Also: IloGoal, IlcOr

Global function IloMaximizeVar

```
public IloSearchSelector IloMaximizeVar(const IloEnv env, const IloNumVar var,
IloNum step=0)
```

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a search selector to act as a filter during the search for a solution in a Concert Technology model.

The search selector that this function creates and returns does several things:

- It stores the leaf of the search tree corresponding to the optimal value of the variable `var` and then reactivates this variable after the complete exploration of the search tree.
- It manages the lower bound on the objective variable. As soon as a solution of value `d` is found, the constraint `var >= d + step` is added to the model for the remainder of the search.
- Open nodes are evaluated. The *evaluation* of an open node is equal to the current maximum of the variable `var` when the node is created. When the search requires an open node, it checks whether the current lower bound on the objective is greater than the evaluation of the node. If so, the node is safely discarded.

This function returns an instance of `IloSearchSelector` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the search selector that it returns as an instance of `IlcSearchSelector` for use during a Solver search.

See Also: IloSearchSelector

Global function IloGenerateBounds

```
public IloGoal IloGenerateBounds(const IloEnv env, const IloNumVar var, IloNum
precision)
public IloGoal IloGenerateBounds(const IloEnv env, const IloNumVarArray vars,
IloNum precision)
```

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal. The goal efficiently reduces the domain of the floating-point variable `var` (or all the domains of all the floating-point variables in the array `vars`) by propagating any constraints on `var` more than usual. It checks whether the boundaries of the domain of `var` are consistent with all the constraints posted on `var`. If that is not the case, then it reduces an interval around `var` until the boundaries become consistent up to the precision indicated by `precision`.

This function works on numerical variables of type `Float` and type `Int`.

Use `precision` to control the effect of this function: if `precision` is small, the new domain computed by `IloGenerateBounds` will be smaller. However, the smaller precision, the longer the computation will take.

When it takes an instance of the class `IloEnv` as a parameter, it returns an instance of `IloGoal` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the goal that it returns as an instance of `IlcGoal` for use during a Solver search.

See Also: IloEnv, IloGoal, IlcGenerateBounds

Global function IlcCos

```
public IlcFloatExp IlcCos(const IlcFloatExp x)
public IlcFloat IlcCos(IlcFloat x)
```

Definition file: ilsolver/nlinflt.h
Include file: <ilsolver/ilosolver.h>

When its argument is a constrained floating-point expression, this function creates a constrained floating-point expression (that is, an instance of `IlcFloatExp` or one of its subclasses) which is equal to the cosine of its argument `x` expressed in radians. The effects of this function are reversible.

When its argument is an unconstrained numeric value (that is, a value of type `IlcFloat`), this function returns the cosine of its argument.

If you want to manipulate constrained floating-point expressions in degrees, we strongly recommend that you call the trigonometric functions on variables expressed in radians and then convert the results to degrees (rather than declaring the constrained floating-point expressions in degrees and then converting them to radians to call the trigonometric functions).

The reason for that advice is that the method we recommend gives more accurate results in the context of the usual floating-point pitfalls. See the *IBM ILOG Solver User's Manual* for an explanation of those pitfalls.

See Also: IlcArcCos, IlcArcSin, IlcArcTan, IlcDegToRad, IlcFloatExp, IlcRadToDeg, IlcSin, IlcTan

Global function IlcNullIntersect

```
public IlcConstraint IlcNullIntersect(IlcAnySetVar a, IlcAnySetVar b)
public IlcConstraint IlcNullIntersect(IlcAnySetVar a, IlcAnySet b)
public IlcConstraint IlcNullIntersect(IlcAnySet a, IlcAnySetVar b)
public IlcConstraint IlcNullIntersect(IlcIntSetVar a, IlcIntSetVar b)
public IlcConstraint IlcNullIntersect(IlcIntSet a, IlcIntSetVar b)
public IlcConstraint IlcNullIntersect(IlcIntSetVar a, IlcIntSet b)
```

Definition file: ilsolver/ilcset.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a constraint. When that constraint is posted, it constrains its two arguments to have an empty intersection in any solution.

See Also: IlcConstraint, IlcAnySetVar, IlcIntSetVar

Global function IlcMinDistance

`public IlcConstraint IlcMinDistance(IlcIntExp x, IlcIntExp y, IlcInt k)`

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

The function `IlcMinDistance` creates and returns a constraint. When that constraint is posted, it insures that the absolute distance between `x` and `y` will be greater than or equal to `k` (that is, `|x - y| >= k`).

`IlcMinDistance(x,y,k)` is more efficient than, although logically equivalent to, the code `IlcAbs(x-y) >= k`.

`IlcAllMinDistance` is a similar function operating on all the variables in an array (rather than on two variables).

See Also: IlcAbs, IlcAllMinDistance

Global function IlcTan

```
public IlcFloatExp IlcTan(const IlcFloatExp x)
public IlcFloat IlcTan(IlcFloat x)
```

Definition file: ilsolver/nlinflt.h
Include file: <ilsolver/ilosolver.h>

When its argument is a constrained floating-point expression, this function creates a constrained floating-point expression (that is, an instance of `IlcFloatExp` or one of its subclasses) which is equal to the tangent of its argument `x` expressed in radians. The effects of this function are reversible.

When its argument is an unconstrained numeric value (that is, a value of type `IlcFloat`), it returns the tangent of its argument expressed in radians.

If you want to manipulate constrained floating-point expressions in degrees, we strongly recommend that you call the trigonometric functions on variables expressed in radians and then convert the results to degrees (rather than declaring the constrained floating-point expressions in degrees and then converting them to radians to call the trigonometric functions).

The reason for that advice is that the method we recommend gives more accurate results in the context of the usual floating-point pitfalls.

See Also: IlcArcCos, IlcArcSin, IlcArcTan, IlcCos, IlcDegToRad, IlcFloatExp, IlcHalfPi, IlcPi, IlcQuarterPi, IlcRadToDeg, IlcSin, IlcThreeHalfPi, IlcTwoPi

Global function IloMonotonicIncreasingNumExpr

`public IloNumExprArg` **`IloMonotonicIncreasingNumExpr`**`(IloNumExprArg node, IloNumFunction f, IloNumFunction invf)`

Definition file: ilconcert/iloexpression.h

For constraint programming: creates a new constrained expression equal to `f(x)`.
This function creates a new constrained expression equal to `f(x)`. The arguments `f` and `invf` must be pointers to functions of type `IlcFloatFunction`. Those two functions must be inverses of one another, that is,

`invf(f(x)) == x` and `f(invf(x)) == x` for all `x`.

`IloMonotonicIncreasingNumExpr` does *not* verify whether `f` and `invf` are inverses of one another. It does *not* verify whether they are monotonically increasing either.

Global function IlcMTBFSEvaluator

```
public IlcNodeEvaluator IlcMTBFSEvaluator(IloSolver solver, IlcIntVar v, IlcInt
step=1)
public IlcNodeEvaluator IlcMTBFSEvaluator(IloSolver solver, IlcFloatVar v, IlcFloat
step=1.0)
```

Definition file: ilsolver/search.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a node evaluator that implements a best first search for a multithreaded search.

Nodes are evaluated according to the variable `v`. As long as the minimum of `v` is no greater than the minimum of the evaluation of each open node `+ s`, the search continues as a depth first search. If the opposite is true, the solver postpones the evaluation of the current node and jumps to the best open node.

If a variable `v` appears in more than one agent (that is, more than one worker), then every copy of the variable `v` must have the same name.

See Also: IlcFloatVarRef, IlcIntVarRef, IlcMTNodeEvaluatorI

Global function IloIfThenElse

```
public IloPredicate< IloObject > IloIfThenElse(IloPredicate< IloObject > pred,
IloPredicate< IloObject > pred1, IloPredicate< IloObject > pred2)
```

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

Creates a conditional predicate.
This function returns a composite predicate which invokes the predicate `pred1` when the predicate `pred` is true and which invokes the predicate `pred2` otherwise.

For more information, see Selectors.

Global function IloIfThenElse

```
public IloEvaluator< IloObject > IloIfThenElse(IloPredicate< IloObject > pred,
IloEvaluator< IloObject > eval1, IloEvaluator< IloObject > eval2)
```

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

Creates a conditional evaluator.
This function returns a composite evaluator whose evaluation is equal to the evaluation of `eval1` in the case where the predicate `pred` is true on the evaluated instance and returns the evaluation of `eval2` otherwise.

For more information, see Selectors.

Global function IloEqMax

```
public IloConstraint IloEqMax(const IloEnv, const IloIntSetVar var1, const
IloIntVar var2, const IloIntToIntFunction f)
public IloConstraint IloEqMax(const IloEnv, const IloIntSetVar var1, const
IloIntVar var2, const IloIntToIntVarFunction f)
```

Definition file: ilconcert/iloset.h

For IBM® ILOG® Solver: a constraint forcing a variable to the maximum of returned values.
This function creates and returns a constraint (an instance of IloConstraint) for use in a model. The constraint forces `var2` to the maximum of the values returned by the function `f` when it is applied to the variable `var1`.

In order for the constraint to take effect, you must add it to a model with the template IloAdd or the member function IloModel::add and extract the model for an algorithm with the member function IloAlgorithm::extract.

Global function IloStoreSolution

```
public IlcGoal IloStoreSolution(IloSolver solver, IloSolution solution)
public IloGoal IloStoreSolution(IloEnv env, IloSolution solution)
```

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function returns a goal which stores `solution`, just as if `solution.store(solver)` had been called. This goal always succeeds.

For more information, see `IloSolution`.

See Also: IloSolution, IloRestoreSolution, IloStoreBestSolution, IloUpdateBestSolution

Global function IlcComputeMin

`public IlcFloat IlcComputeMin(IlcFloat value)`

Definition file: ilsolver/fltexp.h
Include file: <ilsolver/ilosolver.h>

The function returns 0 (zero) when its argument is 0 (zero). When its argument is non-zero, it returns a value of type `IlcFloat` which is less than its argument.

In order to avoid errors due to rounding, Solver uses the technique of outward rounding when working on intervals. During constraint propagation, when new bounds are computed for the domain of a constrained floating-point variable, the newly computed interval is slightly expanded: its lower bound is decreased a little bit, whereas its upper bound is increased a little bit. This practice avoids making intervals smaller than what they would be with exact computation.

The implementation of this technique conforms to the IEEE 754 standard. In this way, Solver provides for consistent and more nearly accurate results in basic arithmetic operations and thus avoids some of the drawbacks of floating-point arithmetic.

Solver uses the function `IlcComputeMin` when it is enlarging an interval of floating-point values by slightly decreasing the lower bound of the interval.

See Also: IlcComputeMax, IlcFloat, IlcFloatMax, IlcFloatMin

Global function IloChooseMinMaxInt

```
public IlcInt IloChooseMinMaxInt(const IlcIntVarArray vars)
```

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the smallest maximum from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseIntIndex, IloGenerate

Global function IloSum

```
public IloNumExprArg IloSum(const IloNumExprArray exprs)
public IloIntExprArg IloSum(const IloIntExprArray exprs)
public IloNumExprArg IloSum(const IloNumVarArray exprs)
public IloIntExprArg IloSum(const IloIntVarArray exprs)
public IloNum IloSum(const IloNumArray values)
public IloInt IloSum(const IloIntArray values)
```

Definition file: ilconcert/iloexpression.h

Returns a numeric value representing the sum of numeric values.
These functions return a numeric value representing the sum of numeric values in the array `vals`, or an instance of `IloNumExprArg`, the internal building block of an expression, representing the sum of the variables in the arrays `exprs` or `values`.

Global function IloAbstraction

```
public IloConstraint IloAbstraction(const IloEnv env, const IloIntVarArray y, const IloIntVarArray x, const IloIntArray values, IloInt abstractValue)
public IloConstraint IloAbstraction(const IloEnv env, const IloAnyVarArray y, const IloAnyVarArray x, const IloAnyArray values, IloAny abstractValue)
```

Definition file: ilconcert/ilomodel.h

For constraint programming: returns a constraint that abstracts the values of one array into the abstract value of another array.
This function returns a constraint that abstracts the values of the elements of one array of constrained variables (called `x`) in a model into the abstract value of another array of constrained variables (called `y`). In other words, for each element `x[i]`, there is a variable `y[i]` corresponding to the abstraction of `x[i]` with respect to an array of numeric `values`. That is,

`x[i] = v` with `v` in `values` if and only if `y[i] = v`;

`x[i] = v` with `v` not in `values` if and only if `y[i] = abstractValue`.

This constraint maintains a many-to-one mapping that makes it possible to define constraints that impinge only on a particular set of values from the domains of constrained variables. The abstract value (specified by `abstractValue`) must not be in the domain of `x[i]`.

Adding a Constraint to a Model, Extracting a Model for an Algorithm

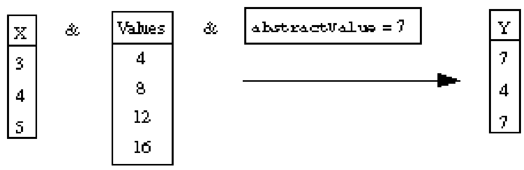
In order for the constraint returned by `IloAbstraction` to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

Exceptions

If the arrays `x` and `y` are not the same size, this function throws the exception `IloAbstraction::InvalidArraysException`.

Example

For simplicity, let's assume that our array `x` consists of three elements with the domains {3}, {4}, and {5}. We will also assume that our abstract value is 7, and the `values` we are interested in are {4, 8, 12, 16}. Then `IloAbstraction` produces these elements in the array `y`:



Global function IlcChooseFirstUnboundBool

`public IlcInt **IlcChooseFirstUnboundBool**(const IlcBoolVarArray vars)`

Definition file: ilsolver/numi.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the first unbound constrained variable that it encounters in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IloGenerate

Global function IlcChooseMaxMaxInt

```
public IlcInt IlcChooseMaxMaxInt(const IlcIntVarArray vars)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the greatest maximum from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseIntIndex, IloGenerate

Global function IlcLinearCtAggregator

```
public IlcConstraintAggregator IlcLinearCtAggregator(IloSolver solver)
```

Definition file: ilsolver/ilosolverhandle.h
Include file: <ilsolver/ilosolver.h>

This aggregator groups linear constraints over integer variables whose bounds are 0 and 1 and propagates faster on the grouped linear constraints.

Global function IloRestartGoal

```
public IloGoal IloRestartGoal(IloEnv env, IloGoal g, IlcInt failLimit, IlcFloat
factor=1.0)
```

Definition file: ilsolver/ilosolverhandle.h
Include file: <ilsolver/ilosolver.h>

The goal returned by this function performs restart on goal `g`. This goal creates a choice point where the left branch calls `g` with a fail limit of `failLimit` and then calls itself on the right branch with `failLimit = failLimit * factor` (with `factor` remaining unchanged). In other words, it runs `g` with a maximum number of fails equal to `failLimit` and then it runs it again with `failLimit * factor` as the new maximum number of fails and so on. For an example of how to use this goal, see *"Using Impacts during Search"* in the *IBM ILOG Solver User's Manual*.

See Also: IlcRestartGoal

Global function IloFirstSolution

```
public IloSearchSelector IloFirstSolution(const IloEnv env, IloInt n=1)
```

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates a search selector that selects the first `n` solutions of a search tree.

When this function takes an instance of the class `IloEnv` as a parameter, it returns an instance of `IloSearchSelector` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the selector that it returns as an instance of `IlcSearchSelector` for use during a Solver search.

See Also: IlcSearchSelector, IloSearchSelector

Global function IlcTableConstraint

```
public IlcConstraint IlcTableConstraint(IlcIntVarArray vars, IlcIntPredicate
predicate)
public IlcConstraint IlcTableConstraint(IlcAnyVarArray vars, IlcAnyPredicate
predicate)
public IlcConstraint IlcTableConstraint(IlcAnyVarArray vars, IlcAnyTupleSet set,
IlcBool compatible)
public IlcConstraint IlcTableConstraint(IlcAnyVar y, IlcConstAnyArray a, IlcIntVar
x)
public IlcConstraint IlcTableConstraint(IlcFloatVar y, IlcConstFloatArray a,
IlcIntVar x)
public IlcConstraint IlcTableConstraint(IlcIntVarArray vars, IlcIntTupleSet set,
IlcBool compatible)
public IlcConstraint IlcTableConstraint(IlcIntVar y, IlcConstIntArray a, IlcIntVar
x)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function can be used to define simple constraints that are not predefined in Solver for use during a Solver search, for example, inside a goal or constraint. (For similar functionality to use in an IBM® ILOG® Concert Technology model, see `IloTableConstraint`.) This function creates and returns a constraint. That constraint is defined

- for *all* the constrained variables in the array `vars`;
- or for the constrained variables `x` and `y`.

The semantics of that generic constraint can be indicated in either one of several ways:

- by a predicate; in that case, the argument `predicate`, of course, indicates that predicate;
- by the values that satisfy the constraint; in that case, the argument `set` indicates the combinations of values that satisfy the constraint, and the argument `compatible` must be `IlcTrue`;
- by the values that do *not* satisfy the constraint; in that case, the argument `set` indicates the unsatisfactory combinations of values, and the argument `compatible` must be `IlcFalse`;
- by making the constrained variable `y` equal to the element of the array `a` at the index indicated by `x`. In other words, `y=a[x];`. This kind of constraint is sometimes known as an *element constraint*.

The order of the constrained variables in the array `vars` is important because the same order is respected in the `predicate` or the `set`. That is, `IlcTableConstraint` passes an array of values to the member function `isTrue` for a predicate or to the member function `isIn` for a set, where the first such value is a value of `vars[0]`, the second is a value of `vars[1]`, and in general, the *ith* value is a value of the constrained variable `vars[i]`.

This function will throw an exception (an instance of `IloSolver::SolverErrorException`) if any of the following conditions occur:

- the function is called with a `predicate` as an argument but the size of the array of constrained variables is greater than three;
- the size of `vars` is different from the size of the `set`.

This function reduces domains efficiently, but it may take some time to do so. The time it needs for domain reduction depends on the size of the domains of the constrained variables in `vars`.

Programming Hint

Solver does not copy the array `a` when `a` is an instance of `IlcConstAnyArray`, `IlcConstFloatArray`, or `IlcConstIntArray`. When a is an instance of one of those constant array classes, then Solver will share the array among constraints instead of copying it. In fact, if you want to build a table constraint of the form `y=a[x]`, we strongly recommend that you should use only the function that takes an instance of `IlcConstFloatArray` or `IlcConstIntArray` as an argument. In that spirit, you can build an instance of `IlcConstIntArray` from an instance of `IlcIntArray`, or an instance of `IlcConstFloatArray` from an instance of `IlcFloatArray`.

Examples:

The following code defines a constraint of arity four such that only these combinations of values are allowed: `(0, 1, 1, 2)`, `(1, 0, 2, 3)`, and `(0, 0, 2, 1)`.

```
IlcIntTupleSet set(s,4);
set.add(IlcIntArray(s,4,0,1,1,2));
set.add(IlcIntArray(s,4,1,0,2,3));
set.add(IlcIntArray(s,4,0,0,2,1));
set.close();

IlcIntVar x(s,0,1),y(s,0,1),z(s,0,3),t(s,0,2);
IlcIntVarArray vars(s,4,x,y,z,t);
```

Inside a goal or constraint now, you can post that constraint by adding it to an instance of `IloSolver`, like this:

```
s.add(IlcTableConstraint(vars,set,IlcTrue));
```

The following code defines a constraint of arity three. It uses a predicate that is true if the three variables are pairwise different or the sum of the first two constrained variables is equal to the third variable.

```
IlcBool IlcIntPredicateI::isTrue(IlcIntArray val) {
   return((val[0] != val[1] && val[1] != val[2] && val[0] != val[2])
               || (val[0] + val[1] == val[2]));
}

IlcIntVar x(s, 0, 3), y(s, 0, 3), z(s, 0, 3);
IlcIntVarArray vars(s, 3, x, y, z);
```

Inside a goal or constraint now, you can post this constraint by adding it to an instance of `IloSolver`, like this:

```
s.add(IlcTableConstraint(vars, Predicate(s));
```

See Also: ILCANYPREDICATE0, IlcAnyTupleSet, IlcConstFloatArray, IlcConstIntArray, IlcConstraint, ILCINTPREDICATE0, IlcIntPredicate, IlcIntTupleSet, IloTableConstraint

Global function IloRestoreSolution

```
public IlcGoal IloRestoreSolution(IloSolver solver, IloSolution solution)
public IloGoal IloRestoreSolution(IloEnv env, IloSolution solution)
```

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function returns a goal to the specified solver that restores `solution`, just as if `solution.restore(solver)` had been called in search mode. If the restoration is legal, according to the constraints of the model, the goal succeeds. Otherwise it fails.

For more information, see `IloSolution`.

See Also: IloSolution, IloStoreBestSolution, IloStoreSolution, IloUpdateBestSolution

Global function IlcSquare

```
public IlcIntExp IlcSquare(const IlcIntExp exp)
public IlcInt IlcSquare(IlcInt i)
public IlcFloatExp IlcSquare(const IlcFloatExp exp)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

When its argument is a constrained expression, this function creates a new constrained expression equal to the square of its argument `exp`. This new expression is logically equivalent to `exp*exp`.

When its argument is an unconstrained numeric value (that is, a value of type `IlcInt`), this function returns the square of its argument.

See Also: IlcExponent, IlcFloatExp, IlcIntExp, IlcPower

Global function operator!

```
public IlcConstraint operator!(const IlcConstraint ct)
```

Definition file: ilsolver/numi.h
Include file: <ilsolver/ilosolver.h>

This operator creates and returns a constraint: the negation of its argument, a constraint.

When you create a constraint, it has no effect until you post it.

See Also: IlcConstraint, operator==, operator<=, operator>=

Global function operator!

`public IloConstraint operator!(const IloConstraint constraint)`

Definition file: ilconcert/ilomodel.h

Overloaded C++ operator for negation.
This overloaded C++ operator returns a constraint that is the negation of its argument. In order to be taken into account, this constraint must be added to a model and extracted by an algorithm, such as `IloCplex` or `IloSolver`.

Global function operator!

`public IloPredicate< IloObject > operator!(IloPredicate< IloObject > pred)`

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

Creates a predicate by negation.
This operator creates a new `IloPredicate<IloObject>` instance from a single `IloPredicate<IloObject>` instance. The semantics of the new instance is a logical `NOT` semantics of the `pred`. That is, the new predicate will return `IloTrue` for a particular object if and only if `pred` returns `IloFalse` for that object.

For more information, see Selectors.

Global function IlcNull

`public IlcConstraint IlcNull(const IlcFloatExp x)`

Definition file: ilsolver/linfloat.h
Include file: <ilsolver/ilosolver.h>

This function is a constraint that forces a constrained floating-point expression to be included in the interval `[-IlcFloatMin, IlcFloatMin]`. It is equivalent to `-IlcFloatMin <= x <= IlcFloatMin`.

This constraint should be used instead of a comparison to 0 (zero). Thus the statement

```
s.add(IlcNull(x)); // Good Practice
```

is better than

```
s.add(x==0.);     // Bad Practice
```

The statement we recommend gives more accurate results in the context of the usual floating-point pitfalls.

See Also: IlcConstraint, IlcFloatExp

Global function IlcLocalImpactVarEvaluator

```
public IloEvaluator< IlcIntVar > IlcLocalImpactVarEvaluator(IloEnv env, IloInt
depth)
public IloEvaluator< IlcIntVar > IlcLocalImpactVarEvaluator(IloSolver solver,
IloInt depth)
```

Definition file: ilsolver/custgoal.h
Include file: <ilsolver/ilosolver.h>

This function returns an instance of the evaluator `IloEvaluator<IlcIntVar>`. The evaluation returns the result of the function `solver.getLocalImpact(x, depth)`, where `x` is the evaluated variable and depth is given in the context.

Global function IloAdd

`public X IloAdd(IloModel & mdl, X x)`

Definition file: ilconcert/ilomodel.h

Template to add elements to a model.
This C++ template helps when you want to add elements to a model. In those synopses, `X` represents a class, `x` is an instance of the class `X`. The class `X` must be `IloExtractable`, `IloExtractableArray`, or one of their subclasses.

If `model` is an instance of `IloModel`, derived from `IloExtractable`, then `x` will be added to the top level of that model.

As an alternative to this way of adding extractable objects to a model, you may also use `IloModel::add`.

This template preserves the original type of its argument `x` when it returns `x`. This feature of the template may be useful, for example, in cases like this:

```
IloRange rng = IloAdd(model, 3 * x + y == 17);
```

For a comparison of these two ways of adding extractable objects to a model, see Adding Extractable Objects in the documentation of `IloExtractable`.

See Also: IloAnd, IloExtractable, IloExtractableArray, IloModel, IloOr

Global function IloOrLimit

`public IloSearchLimit **IloOrLimit**(const IloEnv env, IloInt numOfChoicePts)`

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a search limit. Within this limit, the `solver` explores the search tree for only the number of choice points indicated by `numOfChoicePts`. After that limit has been reached, all remaining unexplored open nodes in the search tree are discarded.

When this function takes an instance of the class `IloEnv` as a parameter, it returns an instance of `IloSearchLimit` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the search limit that it returns as an instance of `IlcSearchLimit` for use during a Solver search.

See Also: IloLimitSearch, IloSearchLimit, IlcSearchLimit

Global function IlcReductionInformation

```
public IlcConstraintAggregator IlcReductionInformation(IloSolver solver)
```

Definition file: ilsolver/ilosolverhandle.h
Include file: <ilsolver/ilosolver.h>

This aggregator maintains information about the reduction rate of the domains of variables. It is required to use the functions:

```
IloInt IloSolver::getReduction(const IlcIntVar x) const;
IloInt IloSolver::getReduction(const IloIntVar x) const;
IloNum IloSolver::getReduction(const IlcFloatVar x) const;
IloNum IloSolver::getReduction(const IloNumVar x) const;
```

See Also: IloSolver::getReduction

Global function IloTimeLimit

```
public IloSearchLimit IloTimeLimit(const IloEnv env, IloNum time)
```

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a search limit. With this limit, the `solver` explores the search tree for only the amount of time indicated by `time` seconds. After that time limit has been reached, all remaining open nodes are discarded.

When this function takes an instance of the class `IloEnv` as a parameter, it returns an instance of `IloSearchLimit` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the search limit that it returns as an instance of `IlcSearchLimit` for use during a Solver search.

See Also: IlcSearchLimit, IloSearchLimit

Global function IlcChooseFirstUnboundInt

`public IlcInt IlcChooseFirstUnboundInt(const IlcIntVarArray vars)`

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the first unbound constrained variable that it encounters in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseIntIndex, IloGenerate

Global function IloSource

`public IloPoolOperator **IloSource**(IloEnv env, IloGoal goal, IloSolution prototype, const char * name=0)`

Definition file: ilsolver/iimoperator.h
Include file: <ilsolver/iim.h>

Creates a solution source from a goal and a solution prototype.
This function creates an operator which executes the supplied goal `goal` and will store solutions according to the prototype `prototype` passed. It is most useful for creating a set of solutions at the head of a processor chain where no input solutions are present to be used as prototypes. `name`, if provided, becomes the name of the new operator.

`IloPoolOperator op = IloSource(goal.getEnv(), goal, prototype)` is short for:

```
IloPoolOperator op(goal);
op.setPrototype(prototype);
```

Note

To produce a variety of solutions, `goal` should be non-deterministic, either by nature or by intentionally perturbing using, for example, `IloRandomPerturbation`.

See Also: IloPoolOperator::IloPoolOperator, IloPoolOperator::setPrototype, IloRandomPerturbation

Global function IlcChooseMinRegretMin

```
public IlcInt IlcChooseMinRegretMin(const IlcIntVarArray vars)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the minimal difference between the minimal possible value and the next minimal possible value from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is based on the principle of *regret*. Regret is the difference between what would have been the best possible decision in a scenario and what was the actual decision.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseIntIndex, IloGenerate

Global function IloBestSolutionComparator

`public IloComparator< IloSolution > **IloBestSolutionComparator**(IloEnv env)`

Definition file: ilsolver/iimpool.h
Include file: <ilsolver/iim.h>

A solution comparator that prefers higher quality solutions.
This function creates a solution comparator on the environment `env` that prefers higher quality solutions. This comparator makes use of `IloSolution::isBetterThan` in order to make its comparison.

See Also: IloWorstSolutionComparator

Global function IloEqUnion

```
public IloConstraint IloEqUnion(const IloEnv env, const IloIntSetVar var1, const
IloIntSetVar var2, const IloIntToIntFunction f)
public IloConstraint IloEqUnion(const IloEnv env, const IloIntSetVar var1, const
IloIntSetVar var2, const IloIntToIntVarFunction f)
public IloConstraint IloEqUnion(const IloEnv env, const IloIntSetVar var1, const
IloIntSetVar var2, const IloIntToIntSetVarFunction f)
```

Definition file: ilconcert/iloset.h

For IBM® ILOG® Solver: a constraint forcing the union of two sets to be the elements of a third set.
This function creates and returns a constraint (an instance of IloConstraint) for use in a model. The constraint forces the union of the values returned by the function `f` when it operates on `var1` to be precisely the elements of the set `var2`.

In order for the constraint to take effect, you must add it to a model with the template IloAdd or the member function IloModel::add and extract the model for an algorithm with the member function IloAlgorithm::extract.

Global function IloEqUnion

```
public IloConstraint IloEqUnion(const IloEnv env, const IloAnySetVar var1, const
IloAnySetVar var2, const IloAnySetVar var3)
public IloConstraint IloEqUnion(const IloEnv env, const IloAnySetVar var, const
IloAnySetVarArray vars)
public IloConstraint IloEqUnion(const IloEnv env, const IloIntSetVar var1, const
IloIntSetVar var2, const IloIntSetVar var3)
public IloConstraint IloEqUnion(const IloEnv, const IloIntSetVar var, const
IloIntSetVarArray vars)
```

Definition file: ilconcert/iloanyset.h

For IBM® ILOG® Solver : a constraint forcing the union of two sets to be the elements of a third set.
This function creates and returns a constraint (an instance of `IloConstraint`) for use in a model. When its arguments are two sets of variables, such as `var2` and `var3`, the constraint forces the union of the sets `var2` and `var3` to be precisely the elements of the set `unionSet`. Likewise, when its arguments include an array of set variables, such as `vars`, the constraint forces the union of the *elements* of that array to be `unionSet`.

In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

Global function IlcPiecewiseLinear

```
public IlcFloatExp IlcPiecewiseLinear(IlcFloatVar x, IlcFloatArray point,
IlcFloatArray slope, IlcFloat a, IlcFloat fa)
```

Definition file: ilsolver/ilcsegfn.h
Include file: <ilsolver/ilosolver.h>

The function `IlcPiecewiseLinear` creates a floating point expression to represent a continuous or discontinuous piecewise linear function *f* of the variable *x*. The array `point` contains the *n* breakpoints of the function such that *point [i-1] ≤ point [i]* for *i = 1, . . ., n-1*. The array `slope` contains the *n+1* slopes of the *n+1* segments of the function. The values `a` and `fa` must be coordinates of a point such that `fa = f(a)`.

When *point[i-1] = point[i]*, there is a step at the x-coordinate point [*i-1],* and its height is slope *[i-1]*.

When appearing in a constraint posted to Solver, this expression always has the smallest possible interval with respect to the interval of x and conversely.

Example

The expression

```
IlcPiecewiseLinear(x, IlcFloatArray(solver, 2, 10., 20.),
                      IlcFloatArray(solver, 3, 0.3, 1., 2.),
                      0, 0);
```

defines a piecewise linear function *f* having two breakpoints at *x = 10* and *x = 20*, and three segments with slopes *0.3*, *1*, and *2*. The first segment has infinite length and ends at the point *x = 10*, *f(x) = 3* since *f(0) = 0*. The second segment starts at the point *x = 10*, *f(x) = 3* and ends at the point *x = 20*, *f(x) = 13* where the third segment starts.

Global function IloDDSEvaluator

`public IloNodeEvaluator` **`IloDDSEvaluator`**`(const IloEnv env, IloInt step=4, IloInt width=2, IloInt maxDiscrepancy=IloIntMax)`

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a node evaluator (an instance of `IloNodeEvaluator`) that implements a variation of Slice-Based Search in a Concert Technology model. (The function `IloSBSEvaluator` returns a node evaluator that implements Slice-Based Search.)

In this variation, discrepancies are confined to the top of the search tree. In the first pass, the goal manager explores nodes with all discrepancies (except `width` discrepancies) appears with a depth less than `step`. In the second pass, it does the same with a depth less than `2*step`, and so on. This variation of slice-based search is more efficient if the search heuristic is very good (that is, if it makes mistakes only in the top of the search tree).

This function returns an instance of `IloNodeEvaluator` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the node evaluator that it returns as an instance of `IlcNodeEvaluator` for use during a Solver search.

See Also: IloApply, IlcNodeEvaluator, IloNodeEvaluator, IloSBSEvaluator

Global function IloCeil

```
public IloEvaluator< IloObject > IloCeil(IloEvaluator< IloObject > eval)
```

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

This function creates a composite `IloEvaluator<IloObject>` instance. This evaluator returns the least integer value not less than the float value returned by the evaluator given as argument.

For more information, see Selectors.

Global function IloCeil

```
public IloNum IloCeil(IloNum val)
```

Definition file: ilconcert/iloenv.h

Returns the least integer value not less than its argument.
This function computes the least integer value not less than `val`.

Examples:

```
IloCeil(IloInfinity) is IloInfinity.
IloCeil(-IloInfinity) is -IloInfinity.
IloCeil(0) is 0.
IloCeil(0.4) is 1.
IloCeil(-0.4) is 0.
IloCeil(0.5) is 1.
IloCeil(-0.5) is 0.
IloCeil(0.6) is 1.
IloCeil(-0.6) is 0.
```

Global function IloImprove

`public IloMetaHeuristic IloImprove(IloEnv env, IloNum step=1e-4)`

Definition file: ilsolver/iimmeta.h
Include file: <ilsolver/iimls.h>

This function takes an environment and an optional step size, and returns an object of type `IloMetaHeuristic` which implements a greedy local search improvement mechanism. If the step size is not supplied, it defaults to 1e-4. When applied to a neighborhood exploration goal, the returned object rejects all movements that do not improve the cost variable by at least `step`.

See Also: IloMetaHeuristic, IloScanDeltas, IloScanNHood, IloSingleMove

Global function IlcChooseMaxRegretMax

```
public IlcInt IlcChooseMaxRegretMax(const IlcIntVarArray vars)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the maximal difference between the maximal possible value and the next maximal possible value from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is based on the principle of *regret*. Regret is the difference between what would have been the best possible decision in a scenario and what was the actual decision.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseIntIndex, IloGenerate

Global function IlcSetMax

```
public IlcGoal IlcSetMax(const IlcIntVar var, const IlcInt val)
public IlcGoal IlcSetMax(const IlcFloatVar var, const IlcFloat val)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function returns a goal which sets the maximum of `var` to be `val`.

See Also: IlcSetMin

Global function IlcEqUnion

```
public IlcConstraint IlcEqUnion(IlcIntSetVar unionset, IlcIntSetVar set,
IlcIntToIntFunction F)
public IlcConstraint IlcEqUnion(IlcIntSetVar unionset, IlcAnySetVar set,
IlcAnyToIntFunction F)
public IlcConstraint IlcEqUnion(IlcAnySetVar unionset, IlcIntSetVar set,
IlcIntToAnyFunction F)
public IlcConstraint IlcEqUnion(IlcAnySetVar unionset, IlcAnySetVar set,
IlcAnyToAnyFunction F)
```

Definition file: ilsolver/setcst.h
Include file: <ilsolver/ilosolver.h>

These functions create and return a constraint that forces the variable `unionset` to be equal to the union of the values returned by the function `F` when applied to the elements of the constrained set variable `set`.

Adding These Constraints

You may add these constraints only during a Solver search; that is, inside a goal (an instance of `IlcGoal`) or inside a constraint (an instance of `IlcConstraint`). If you are looking for similar functionality in a constraint to add to a *model*, see `IloEqUnion`.

Example

It can be useful to constrain the values of an attribute of a computed set of objects. For example, if we have to assign crew members to a flight, and if each crew member has an attribute that describes the language he or she speaks, with this `IlcEqUnion` constraint, it is possible to post constraints on the set of languages that must be spoken during a flight.

```
enum Language {English, French, German};
class CrewMember {
public:
   const char* _name;
   IlcInt      _language;
   CrewMember(IloSolver s, const char* name, Language lang);
};
//Access to the language spoken by a crew member
IlcAnyToIntFunction languages;
//Possible crew members
IlcAnyArray c(s, 3);
c[0]= new (s.getHeap()) CrewMember(s, "John", English);
c[1]= new (s.getHeap()) CrewMember(s, "Kai", German);
c[2]= new (s.getHeap()) CrewMember(s, "Julie", French);
//The flight
IlcAnySetVar crew(s, c, "NewYork-Paris");
//The languages spoken on this flight
IlcIntSetVar langs(s, IlcIntArray(s, 3, English, French, German));
```

Now that we have defined the classes and constraints, we add them during a Solver search, for example, inside a goal or constraint.

```
s.add(IlcEqUnion(langs, crew, languages));
//At least 2 different languages spoken
s.add(IlcCard(langs) >= 2);
//French must be spoken
s.add(IlcMember(French, langs));
```

See Also: IlcAnySetVar, IlcIntSetVar, IlcUnion, IloEqUnion

Global function IlcEqUnion

```
public IlcConstraint IlcEqUnion(IlcIntSetVar unionset, IlcIntSetVar set,
IlcIntToIntExpFunction F)
public IlcConstraint IlcEqUnion(IlcIntSetVar unionset, IlcAnySetVar set,
IlcAnyToIntExpFunction F)
public IlcConstraint IlcEqUnion(IlcAnySetVar unionset, IlcIntSetVar set,
IlcIntToAnyExpFunction F)
public IlcConstraint IlcEqUnion(IlcAnySetVar unionset, IlcAnySetVar set,
IlcAnyToAnyExpFunction F)
```

Definition file: ilsolver/setcst.h
Include file: <ilsolver/ilosolver.h>

These functions create and return a constraint that forces the variable `unionset` to be equal to the union of the values returned by the function `F` when applied to the elements of the constrained set variable `set`. The values returned by `F` are constrained expressions or variables (that is, instances of `IlcIntExp`, `IlcIntVar`, `IlcAnyExp`, or `IlcAnyVar`).

Adding These Constraints

You may add these constraints only during a Solver search; that is, inside a goal (an instance of `IlcGoal`) or inside a constraint (an instance of `IlcConstraint`). If you are looking for similar functionality in a constraint to add to a *model*, see `IloEqUnion`.

Example

These `IlcEqUnion` constraints can be useful to express constraints on the values of a constrained attribute of a computed set of objects. For example, if we have to connect cards to a rack, and if for each card we also have to connect one sensor, it is possible to express constraints on the set of sensors connected to the cards in the rack. To do so, we define the following classes.

```
class Sensor {
public:
   const char* _name;
   Sensor(IloSolver s, const char* name);
};
class Card {
public:
   IlcAnyVar _sensor;
   Card(IloSolver s, IlcAnyArray sensors)
    :_sensor(s, sensors) {}
};
//Access to the sensor connected to a card
IlcAnyToAnyExpFunction sensorsAccess;
//The possible sensors
IlcAnyArray sensors(s, 4);
sensors[0] = new (s.getHeap()) Sensor(s, "Sensor#0");
sensors[1] = new (s.getHeap()) Sensor(s, "Sensor#1");
sensors[2] = new (s.getHeap()) Sensor(s, "Sensor#2");
sensors[3] = new (s.getHeap()) Sensor(s, "Sensor#3");
//The possible cards
IlcAnyArray cards(s, 3);
cards[0] = new (s.getHeap()) Card(s, sensors);
cards[1] = new (s.getHeap()) Card(s, sensors);
cards[2] = new (s.getHeap()) Card(s, sensors);
//The cards connected to the rack
IlcAnySetVar rackCards(s, cards, "Rack#1");
//The sensors connected to the rack
IlcAnySetVar rackSensors(s, sensors);
```

Now that we have defined the classes and constraints, we add them during a Solver search, for example, inside a goal or constraint.

```
s.add(IlcEqUnion(rackSensors, rackCards, sensorAccess));
//At most 10 sensors in the rack
s.add(IlcCard(rackSensors) <= 10);
```

```
//sensor#1 in a card of the rack
s.add(IlcMember(sensors[1], rackSensors));
```

Of course it is possible to compose several levels of indirection. For example, we can post constraints on the processes assigned to the sensors which are connected to the cards that are plugged into a rack:

```
s.add(IlcEqUnion(rackSensors,   rackCards,   sensorAccess));
s.add(IlcEqUnion(rackProcesses, rackSensors, processAccess));
```

See Also: IlcAnySetVar, IlcIntSetVar, IlcUnion, IloEqUnion

Global function IlcEqUnion

```
public IlcConstraint IlcEqUnion(IlcIntSetVar unionset, IlcIntSetVar var1,
IlcIntSetVar var2)
public IlcConstraint IlcEqUnion(IlcAnySetVar unionset, IlcAnySetVar var1,
IlcAnySetVar var2, IlcFilterLevel level)
public IlcConstraint IlcEqUnion(IlcAnySetVar unionset, IlcAnySetVar var1,
IlcAnySetVar var2)
public IlcConstraint IlcEqUnion(IlcAnySetVar unionset, IlcAnySetVar var1,
IlcAnySetVar var2, IlcAnySetVar intersection)
public IlcConstraint IlcEqUnion(IlcIntSetVar unionset, IlcIntSetVar var1,
IlcIntSetVar var2, IlcFilterLevel level)
public IlcConstraint IlcEqUnion(IlcIntSetVar unionset, IlcIntSetVar var1,
IlcIntSetVar var2, IlcIntSetVar intersection)
public IlcConstraint IlcEqUnion(IlcIntSetVar unionset, IlcIntSetVarArray vars)
public IlcConstraint IlcEqUnion(IlcAnySetVar unionset, IlcAnySetVarArray vars)
```

Definition file: ilsolver/intset.h
Include file: <ilsolver/ilosolver.h>

These functions create and return a constraint that forces the value of `unionset` to be equal to the union of its other parameters (that is, the union of the set variables `var1` and `var2` or the union of the set variables in the array `vars`). The variable `unionset` and all the variables in `vars` must be built from the same initial array.

When you pass this function the optional parameter `intersection`, you enable Solver to propagate at a stronger filter level. It takes into account the fact that the cardinality of the `unionset` is equal to the cardinality of `var1` plus the cardinality of `var2` minus the cardinality of their `intersection`.

When you pass this function the optional parameter `level`, it computes the intersection of `var1` and `var2` internally and uses that information to extend the filter level during propagation. See the enumeration `IlcFilterLevel` for more details about its use.

Adding These Constraints

You may add these constraints only during a Solver search; that is, inside a goal (an instance of `IlcGoal`) or inside a constraint (an instance of `IlcConstraint`). If you are looking for similar functionality in a constraint to add to a *model*, see `IloEqUnion` documented in the *Concert Technology Reference Manual*.

See Also: IlcAnySetVar, IlcAnySetVarArray, IlcConstraint, IlcFilterLevel, IlcIntSetVar, IlcIntSetVarArray, IlcUnion

Global function IlcEqUnion

```
public IlcConstraint IlcEqUnion(IlcIntSetVar unionset, IlcIntSetVar set,
IlcIntToIntSetVarFunction F)
public IlcConstraint IlcEqUnion(IlcIntSetVar unionset, IlcAnySetVar set,
IlcAnyToIntSetVarFunction F)
public IlcConstraint IlcEqUnion(IlcAnySetVar unionset, IlcIntSetVar set,
IlcIntToAnySetVarFunction F)
public IlcConstraint IlcEqUnion(IlcAnySetVar unionset, IlcAnySetVar set,
IlcAnyToAnySetVarFunction F)
```

Definition file: ilsolver/setcst.h
Include file: <ilsolver/ilosolver.h>

These functions create and return a constraint that forces the variable `unionset` to be equal to the union of the values returned by the function `F` when applied to the elements of the constrained set variable `set`. The values returned by `F` are constrained set variables (that is, instances of `IlcInt`SetVar or `IlcAny`SetVar).

Adding These Constraints

You may add these constraints only during a Solver search; that is, inside a goal (an instance of `IlcGoal`) or inside a constraint (an instance of `IlcConstraint`). If you are looking for similar functionality in a constraint to add to a *model*, see `IloEqUnion`.

Example

These `IlcEqUnion` constraints can be useful to express constraints on the values of a constrained set attribute of a computed set of objects. For example, if we have to connect cards to a rack, and if for each card we also have to connect a set of sensors, it is possible to express constraints on the set of sensors connected to the cards in the rack.

```
class Sensor {
public:
   const char* _name;
   Sensor(IloSolver s, const char* name);
};
class Card {
public:
   IlcAnySetVar _sensor;
   Card(IloSolver s, IlcAnyArray sensors)
    :_sensor(s, sensors) {}
};
//Access to the sensor connected to a card
IlcAnyToAnySetVarFunction sensorsAccess;
//The possible sensors
IlcAnyArray sensors(s, 4);
sensors[0] = new (s.getHeap()) Sensor(s, "Sensor#0");
sensors[1] = new (s.getHeap()) Sensor(s, "Sensor#1");
sensors[2] = new (s.getHeap()) Sensor(s, "Sensor#2");
sensors[3] = new (s.getHeap()) Sensor(s, "Sensor#3");
//The possible cards
IlcAnyArray cards(s, 3);
cards[0] = new (s.getHeap()) Card(s, sensors);
cards[1] = new (s.getHeap()) Card(s, sensors);
cards[2] = new (s.getHeap()) Card(s, sensors);
//The cards connected to the rack
IlcAnySetVar rackCards(s, cards, "Rack#1");
//The sensors connected to the rack
IlcAnySetVar rackSensors(s, sensors);
```

Now that we have defined these classes and constraints, we add them during a Solver search, for example, inside a goal or constraint.

```
s.add(IlcEqUnion(rackSensors, rackCards, sensorAccess));
//At most 10 sensors in the rack
s.add(IlcCard(rackSensors) <= 10);
//sensor#1 in a card of the rack
```

```
s.add(IlcMember(sensors[1], rackSensors));
```

Of course it is possible to compose several levels of indirection. For example, we can access the processes assigned to the sensors which are connected to the cards that are plugged into a rack:

```
s.add(IlcEqUnion(rackSensors,   rackCards,   sensorAccess));
s.add(IlcEqUnion(rackProcesses, rackSensors, processAccess));
```

See Also: IlcAnySetVar, IlcIntSetVar, IlcUnion, IloEqUnion

Global function ILORTTIN

`public ILORTTIN(ILOTEMPLATECLASS_2, (IloDefaultVisitorI, IloObject, IloContainer), ILOTEMPLATECLASS_2, (IloVisitorI, IloObject, IloContainer), ILOGENTEMPLATE_2, (class IloObject, class IloContainer))`

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

This macro allows you to define a default visitor for a given object class `tx` and a given container class `tc`.

Within the code of this macro, the function `void visit(tx object)` allows you to specify each visited object.

Here is an example of the actual definition of the default visitor `IloVisitor<IloIntVar,IloIntVarArray>`:

```
ILODEFAULTVISITOR(IloIntVar,IloIntVarArray,array) {
   const IloInt size = array.getSize();
   for (IloInt i=0; i<size; ++i)
       visit(array[i]);
}
```

The following default visitors are already defined in Solver:

- `<IloInt,IloIntArray>`
- `<IloNum,IloNumArray>`
- `<IloBool,IloBoolArray>`
- `<IloIntVar,IloIntVarArray>`
- `<IloNumVar,IloNumVarArray>`
- `<IloBoolVar,IloBoolVarArray>`
- `<IlcIntVar,IlcIntVarArray>`
- `<IlcFloatVar,IlcFloatVarArray>`
- `<IlcInt,IlcIntArray>`
- `<IlcFloat,IlcFloatArray>`

For more information, see Selectors.

See Also: IloBestSelector, IloVisitor, ILOVISITOR0

Global function IloIntersection

```
public IloNumToAnySetStepFunction IloIntersection(const IloNumToAnySetStepFunction
f1, const IloNumToAnySetStepFunction f2)
```

Definition file: ilconcert/ilosetfunc.h

creates and returns a function equal to the intersection between the functions.
This operator creates and returns a function equal to the intersection between the functions `f1` and `f2`. The argument functions `f1` and `f2` must be defined on the same interval. The resulting function is defined on the same interval as the arguments. See also: `IloNumToAnySetStepFunction`.

Global function operator%

```
public IloIntExprArg operator%(IloInt x, const IloIntExprArg y)
```

Definition file: ilconcert/iloexpression.h

Returns an expression equal to the modulo of its arguments.
This operator returns an instance of `IloIntExprArg`, the internal building block of an expression, representing the modulo of the integer value `x` and the expression `y`.

Global function operator%

```
public IloIntExprArg operator%(const IloIntExprArg x, IloInt y)
```

Definition file: ilconcert/iloexpression.h

Returns an expression equal to the modulo of its arguments.
This operator returns an instance of `IloIntExprArg`, the internal building block of an expression, representing the modulo of the expression `x` and the integer value `y`.

Global function IlcComputeMax

```
public IlcFloat IlcComputeMax(IlcFloat value)
```

Definition file: ilsolver/fltexp.h
Include file: <ilsolver/ilosolver.h>

The function returns 0 (zero) when its argument is 0 (zero). When its argument is non-zero, it returns a value of type `IlcFloat` which is greater than its argument.

In order to avoid errors due to rounding, Solver uses the technique of outward rounding when working on intervals. During constraint propagation, when new bounds are computed for the domain of a constrained floating-point variable, the newly computed interval is slightly expanded: its lower bound is decreased a little bit, whereas its upper bound is increased a little bit. This practice avoids making intervals smaller than what they would be with exact computation.

The implementation of this technique conforms to the IEEE 754 standard for floating-point arithmetic. In this way, Solver provides for consistent and more nearly accurate results in basic arithmetic operations and thus avoids some of the drawbacks of floating-point arithmetic.

Solver uses the function `IlcComputeMax` when it is enlarging an interval of floating-point values by slightly increasing the upper bound of the interval.

See Also: IlcComputeMin, IlcFloat, IlcFloatMax, IlcFloatMin

Global function IlcAbstraction

```
public IlcIntVarArray IlcAbstraction(IlcIntVarArray vars, IlcIntArray values,
IlcInt abstractValue)
public IlcAnyVarArray IlcAbstraction(IlcAnyVarArray vars, IlcAnyArray values,
IlcInt abstractValue)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns an array of constrained variables for use during a Solver search. The argument `vars` should be an array of constrained variables. The argument `values` is an array of integers or pointers. The argument `abstractValue` is a value that must *not* belong to any of the variables in `vars` and must not appear in `values`.

For each `vars[i]`, Solver creates a variable corresponding to the *abstraction* of `vars[i]` with respect to `values`. In other words, for every variable in `vars`, `vars[i]`, Solver creates a variable `w[i]` such that the domain of `w[i]` is equal to the set made up of the value `abstractValue` plus all those values `vars[j]` that also belong to the array `values`. Then internally, Solver enforces these conditions:

- `vars[i]` is one of the values in the array `values` if and only if `w[i] == vars[i]`;
- `vars[i]` is *not* one of the values in the array `values` if and only if `w[i] == abstractValue`.

This function makes it easy to define constraints that impinge only on a particular set of values from the domains of constrained variables.

For a function that returns a constraint (rather than an array), see `IlcEqAbstraction`.

For a constraint suitable for use in a *model*, see `IloAbstraction`.

See Also: IloAbstraction, IlcBoolAbstraction, IlcEqAbstraction

Global function IloMinimize

```
public IloObjective IloMinimize(const IloEnv env, IloNum constant=0.0, const char * name=0)
public IloObjective IloMinimize(const IloEnv env, const IloNumExprArg expr, const char * name=0)
```

Definition file: ilconcert/ilolinear.h

Defines a minimization objective.
This function defines a minimization objective in a model. In other words, it simply offers a convenient way to create an instance of `IloObjective` with its sense defined as `Minimize`. However, an instance of `IloObjective` created by `IloMinimize` may not necessarily maintain its sense throughout the lifetime of the instance. The optional argument `name` is set to `0` by default.

You may define more than one objective in a model. However, algorithms conventionally take into account only one objective at a time.

Global function IlcSum

```
public IlcIntExp IlcSum(const IlcIntSetVar setVar, const IlcIntToIntFunction F)
public IlcIntExp IlcSum(const IlcIntSetVar setVar, const IlcIntToIntExpFunction F)
public IlcIntExp IlcSum(const IlcAnySetVar setVar, const IlcAnyToIntFunction F)
public IlcIntExp IlcSum(const IlcAnySetVar setVar, const IlcAnyToIntExpFunction F)
```

Definition file: ilsolver/setcst.h
Include file: <ilsolver/ilosolver.h>

These functions create and return a new constrained expression equal to the sum of the values returned by the function `F` applied to the elements assigned to the constrained set variable `setVar`.

$$y = \sum_{x \in setVar} F(x)$$

The value returned by the function `F` can be an integer value (`IlcInt`) or a constrained integer expression (an instance of `IlcIntExp`).

The minimum value of the expression is a sum of two sums, one based on the function `F` applied to *required* elements of `setVar`, and the other based on the function `F` applied to the *possible* elements of `setVar`. The minimum is calculated in this way:

$$\left(\sum_{i \in required(setVar)} min(F(i)) \geq 0\right) + \left(\sum_{i \in possible(setVar)} min(F(i)) < 0\right)$$

Likewise, the maximum value of the returned expression is a sum of two sums, one based on `F` applied to the *possible* elements of `setVar` and the other sum based on `F` applied to *required* elements of `setVar`.

$$\left(\sum_{i \in possible(setVar)} max(F(i)) \geq 0\right) + \left(\sum_{i \in required(setVar)} max(F(i)) < 0\right)$$

The bounds of the expression also depend on the cardinality variable of `setVar`.

See Also: IlcAnySetVar, IlcIntExp, IlcIntSetVar

Global function IlcSum

```
public IlcFloatExp IlcSum(const IlcIntSetVar aSet, const IlcIntToFloatExpFunction
F)
```

Definition file: ilsolver/setcst.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a floating-point expression constrained to be the sum of the floating-point expressions returned by the function `F` over its domain, `aSet`.

See Also: IlcIntSetVar

Global function IlcSum

```
public IlcIntExp IlcSum(const IlcIntSetVar setVar)
```

Definition file: ilsolver/setcst.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a new constrained expression equal to the sum of the elements that are assigned to the constrained set variable `setVar`. The set may contain positive or negative integers.

$$y = \sum_{x \in \mathrm{setVar}} x$$

The minimum value of the expression is the sum of the required elements of the variable `setVar`.

The maximum value of the expression is the sum of the possible elements of the variable `setVar`.

The bounds of the expression also depend on the cardinality variable of `setVar`.

See Also: IlcIntExp, IlcIntSetVar

Global function IlcSum

```
public IlcFloatExp IlcSum(const IlcFloatVarArray vars)
public IlcIntExp IlcSum(const IlcIntVarArray array)
public IlcInt IlcSum(const IlcIntArray array)
public IlcFloat IlcSum(const IlcFloatArray array)
```

Definition file: ilsolver/linfloat.h
Include file: <ilsolver/ilosolver.h>

This function creates a new constrained expression equal to the sum of the constrained expressions in `array`, an array of constrained values or variables that may be integer or floating-point.

When its argument is an instance of `IlcIntArray` or `IlcFloatArray`, it simply creates and returns the sum of the elements.

When its argument is an array of constrained integer variables, then its domain is an interval. Its minimum is the sum of the minimums of the elements of `array`. Its maximum is the sum of the maximums of the elements of `array`.

See Also: IlcFloatExp, IlcFloatVarArray, IlcIntExp, IlcIntVarArray, IlcScalProd

Global function operator+

```
public IloMetaHeuristic operator+(IloMetaHeuristic mh1, IloMetaHeuristic mh2)
```

Definition file: ilsolver/iimmeta.h
Include file: <ilsolver/iimls.h>

This operator creates a *combined metaheuristic*. A combined meta-heuristic filters moves at least as strongly as either `mh1` or `mh2` alone. In fact, a proposed move is rejected by the resulting metaheuristic if *either*`mh1` or `mh2` would reject the move.

More specficially, for the newly created metaheuristic, all the following member functions call the corresponding member functions on `mh1` and `mh2` with the same parameters: `reset`, `start`, `isFeasible`, `test`, `notify`, `complete`.

With respect to return values, `start`, `isFeasible` and `test` return a true value if and only if both `mh1` and `mh2` return true values. `complete` returns a true value if and only if one or both of `mh1` and `mh2` return a true value.

See Also: IloMetaHeuristic, IloTest, IloStart, IloSolutionDeltaCheck

Global function operator+

```
public IlcFloatExp operator+(const IlcFloatExp exp1, IlcFloat exp2)
public IlcIntExp operator+(const IlcIntExp exp1, IlcInt exp2)
public IlcIntExp operator+(IlcInt exp1, const IlcIntExp exp2)
public IlcIntExp operator+(const IlcIntExp exp1, const IlcIntExp exp2)
public IlcFloatExp operator+(IlcFloat exp1, const IlcFloatExp exp2)
public IlcFloatExp operator+(const IlcFloatExp exp1, const IlcFloatExp exp2)
public IlcIntToIntStepFunction operator+(const IlcIntToIntStepFunction & f1, const
IlcIntToIntStepFunction & f2)
```

Definition file: ilsolver/linfloat.h
Include file: <ilsolver/ilosolver.h>

This arithmetic operator adds its arguments. It has been overloaded to handle constrained expressions appropriately. The domain of the resulting expression is computed from the domains of the combined expressions as you would expect. For example, the domain of `x + y` is composed of all the sums of `a + b` where `a` ranges over the domain of `x` and `b` ranges over the domain of `y`.

See Also: IlcFloatExp, IlcIntExp, IlcIntToIntStepFunction

Global function operator+

```
public IloNumToNumStepFunction operator+(const IloNumToNumStepFunction f1, const
IloNumToNumStepFunction f2)
```

Definition file: ilconcert/ilonumfunc.h

Creates and returns a function equal the sum of its argument functions.
This operator creates and returns a function equal the sum of the functions `f1` and `f2`. The argument functions `f1` and `f2` must be defined on the same interval. The resulting function is defined on the same interval as the arguments. See also: `IloNumToNumStepFunction`.

Global function operator+

```
public IloNHood operator+(IloNHood nhood1, IloNHood nhood2)
```

Definition file: ilsolver/iimnhood.h

This operator creates a *concatenated neighborhood* and is shorthand for the following code (assuming the result is placed in `nhood`):

```
IloEnv env = nhoods.getEnv();
IloNHoodArray nhoods(env, 2);
nhoods[0] = nhood1;
nhoods[1] = nhood2;
nhood = IloConcatenate(env, nhoods);
```

See Also: IloConcatenate, IloNHood

Global function operator+

```
public IloNumExprArg operator+(const IloNumExprArg x, const IloNumExprArg y)
public IloNumExprArg operator+(const IloNumExprArg x, IloNum y)
public IloNumExprArg operator+(IloNum x, const IloNumExprArg y)
public IloIntExprArg operator+(const IloIntExprArg x, const IloIntExprArg y)
public IloIntExprArg operator+(const IloIntExprArg x, IloInt y)
public IloIntExprArg operator+(IloInt x, const IloIntExprArg y)
```

Definition file: ilconcert/iloexpression.h

Returns an expression equal to the sum of its arguments.
This overloaded C++ operator returns an expression equal to the sum of its arguments. Its arguments may be numeric values, numeric variables, or other expressions.

Global function operator+

```
public IloEvaluator< IloObject > operator+(IloEvaluator< IloObject > left,
IloEvaluator< IloObject > right)
public IloEvaluator< IloObject > operator+(IloEvaluator< IloObject > left, IloNum
c)
public IloEvaluator< IloObject > operator+(IloNum c, IloEvaluator< IloObject >
right)
```

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

These operators create a composite `IloEvaluator<IloObject>` instance. The semantics of the new evaluator are the addition of the values of the component evaluators. The first function combines two evaluators, adding their values to generate the combined evaluation. The other two signatures add the value returned by the evaluator with an `IloNum` value.

For more information, see Selectors.

Global function IloTest

```
public IloGoal IloTest(IloEnv env, IloMetaHeuristic mh)
public IloGoal IloTest(IloEnv env, IloMetaHeuristic mh, IloNeighborIdentifier nid)
public IlcGoal IloTest(IloSolver solver, IloMetaHeuristic mh, IlcNeighborIdentifier
nid)
public IlcGoal IloTest(IloSolver solver, IloMetaHeuristic mh)
```

Definition file: ilsolver/iimls.h
Include file: <ilsolver/iimls.h>

These functions return a goal which tests if the current variable instantiation is consistent with a metaheuristic.

If specified, `nid` is used to communicate the index and the delta of a neighbor supplied by goals `IloScanDeltas` and `IloScanNHood`.

Conversely, when no delta is specified, these functions return a goal that calls `mh.test(solver, IloSolution())`. If the testing method does not cause a failure and returns `IloTrue`, the goal succeeds. Otherwise it fails.

See Also: IloApplyMetaHeuristic, IloMetaHeuristic, IloNotify, IloScanDeltas, IloScanNHood, IloSingleMove, IloStart

Global function IloIsNAN

```
public int IloIsNAN(double)
```

Definition file: ilconcert/ilosys.h

Tests whether a double value is a NaN.
This function tests whether a double value is a NaN (Not a number).

Global function IloChooseMinRegretMin

`public IlcInt **IloChooseMinRegretMin**(const IlcIntVarArray vars)`

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the minimal difference between the minimal possible value and the next minimal possible value from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is based on the principle of *regret*. Regret is the difference between what would have been the best possible decision in a scenario and what was the actual decision.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseIntIndex, IloGenerate

Global function IlcOr

```
public IlcGoal IlcOr(const IlcGoal g1, const IlcGoal g2, IlcAny label=0)
public IlcGoal IlcOr(const IlcGoal g1, const IlcGoal g2, const IlcGoal g3, IlcAny
label=0)
public IlcGoal IlcOr(const IlcGoal g1, const IlcGoal g2, const IlcGoal g3, const
IlcGoal g4, IlcAny label=0)
public IlcGoal IlcOr(const IlcGoal g1, const IlcGoal g2, const IlcGoal g3, const
IlcGoal g4, const IlcGoal g5, IlcAny label=0)
```

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>

A goal can be defined as a choice between other goals. The function `IlcOr` implements a goal as a choice between subgoals (between two and five subgoals). When this goal is executed, the state of Solver, including the state of the goal stack, is saved. This activity is called *setting a choice point*. Then all the subgoals are saved as untried subgoals of the choice point. Then the first untried subgoal is removed from the set of untried subgoals of the choice point and is called. If it fails, the state of Solver is restored and the next untried subgoal is called, and so on, until either a subgoal succeeds or until no more untried subgoals remain. In the latter case, the choice point itself fails. If the optional argument, `label`, is given, the choice point is labeled with it. The apparent limitation to five subgoals can be overcome by several calls to `IlcOr` since it is associative.

If a goal is null (that is, if its implementation is null), it will be silently transformed into a goal that always succeeds.

Examples:

For example, the following goal has three choices:

```
ILCGOAL0(PrintOne) {
    IloSolver s = getSolver();
    s.out() << ?print one? << endl;
    return IlcOr(PrintX(s, 1), PrintX(s, 2), PrintX(s, 3)));
}
```

Here is how to define a choice point with eight subgoals:

```
IlcOr(IlcOr(g1, g2, g3, g4, g5),
            IlcOr(g6, g7, g8));
```

For more information, see the concepts Goal and Choice Point.

See Also: IlcAnd, IlcGoal

Global function IlcMonotonicIncreasingFloatExp

`public IlcFloatExp` **`IlcMonotonicIncreasingFloatExp`**`(IlcFloatExp x, IlcFloatFunction f, IlcFloatFunction invf, const char * functionName=0)`

Definition file: ilsolver/nlinflt.h
Include file: <ilsolver/ilosolver.h>

This function creates a new constrained expression equal `f(x)`. The arguments `f` and `invf` must be pointers to functions of type `IlcFloatFunction`. Those two functions must be inverses of one another, that is,

`invf(f(x)) == x` and `f(invf(x)) == x` for all `x`.

Those two functions must also be monotonically increasing.

`IlcMonotonicIncreasingFloatExp` does *not* verify whether `f` and `invf` are inverses of one another. It does *not* verify whether they are monotonically increasing either.

The effects of this function are reversible.

Examples:

Here's how to define `IlcExponent` and `IlcLog` in terms of this function:

```
static IlcFloat CallFloatExponent(IlcFloat x) {
    return (x > 709) ?  1e308 : exp(x);
}
static IlcFloat CallFloatLog(IlcFloat x) {
    return  (x <= 0) ? -1e308 : log(x);
}
IlcFloatExp IlcExponent(const IlcFloatExp var) {
  return IlcMonotonicIncreasingFloatExp
            (var, CallFloatExponent,CallFloatLog, ?IlcExponent?);
}
IlcFloatExp IlcLog(const IlcFloatExp var) {
  return IlcMonotonicIncreasingFloatExp
            (var, CallFloatLog,CallFloatExponent, ?IlcLog?);
}
```

See Also: IlcExponent, IlcFloatExp, IlcFloatFunction, IlcLog, IlcMonotonicDecreasingFloatExp, IlcPower, IlcSquare

Global function IlcArcTan

```
public IlcFloatExp IlcArcTan(const IlcFloatExp x)
public IlcFloat IlcArcTan(IlcFloat x)
```

Definition file: ilsolver/nlinflt.h
Include file: <ilsolver/ilosolver.h>

When its argument is a constrained floating-point expression, this function creates a constrained floating-point expression (that is, an instance of `IlcFloatExp` or one of its subclasses) which is equal to the arc tangent (in the range -Pi/2 to Pi/2) of its argument `x` expressed in radians. The effects of this function are reversible.

When its argument is an unconstrained numeric value (that is, a value of type `IlcFloat`), it returns the arc tangent of its argument expressed in radians.

If you want to manipulate constrained floating-point expressions in degrees, we strongly recommend that you call the trigonometric functions on variables expressed in radians and then convert the results to degrees (rather than declaring the constrained floating-point expressions in degrees and then converting them to radians to call the trigonometric functions).

The reason for that advice is that the method we recommend gives more accurate results in the context of the usual floating-point pitfalls.

See Also: IlcArcCos, IlcArcSin, IlcCos, IlcDegToRad, IlcFloatExp, IlcHalfPi, IlcRadToDeg, IlcPi, IlcQuarterPi, IlcSin, IlcTan, IlcThreeHalfPi, IlcTwoPi

Global function IlcChooseMinMaxFloat

```
public IlcInt IlcChooseMinMaxFloat(const IlcFloatVarArray vars)
```

Definition file: ilsolver/linfloat.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the smallest maximum from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcFloatVarArray, IloGenerate

Global function IlcSuccessRateValueEvaluator

```
public IloEvaluator< IlcInt > IlcSuccessRateValueEvaluator(IloEnv env)
public IloEvaluator< IlcInt > IlcSuccessRateValueEvaluator(IloSolver solver)
```

Definition file: ilsolver/custgoal.h
Include file: <ilsolver/ilosolver.h>

This function returns an instance of the evaluator `IloEvaluator<IlcInt>`. The evaluation returns the result of the function `solver.getSuccessRate(x, a)`, where `x` is the selected variable given in context and `a` is the value evaluated.

Global function IloCommitDelta

```
public IlcGoal IloCommitDelta(IloSolver solver, IloSolution solution, IloSolution
delta)
public IloGoal IloCommitDelta(IloEnv env, IloSolution solution, IloSolution delta)
```

Definition file: ilsolver/iimls.h
Include file: <ilsolver/iimls.h>

This function takes solution `solution` and delta `delta`, and returns a goal for the specified solver that:

1. Restores `delta`.
2. Restores the part of `solution` that does not appear in `delta`.
3. Saves the resulting solution. The resulting solution consists of `solution` as modified by `delta`.

The goal can fail at 1) or 2) if any constraints are violated, that is, the goal fails if the proposed change to the solution is illegal.

Implementation

`IloCommitDelta` can be implemented as:

```
IlcGoal IloCommitDelta(solver, solution, delta) {
  return IlcAnd(IloTestDelta(solver, solution, delta),
                IloStoreSolution(solver, solution));
}
```

For more information, see `IloSolution` in the *Concert Technology Reference Manual*.

See Also: IloStoreSolution, IloTestDelta

Global function IlcChooseFirstUnboundAny

```
public IlcInt IlcChooseFirstUnboundAny(const IlcAnyVarArray vars)
```

Definition file: ilsolver/ilcany.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the first unbound constrained variable that it encounters in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseAnyIndex, IloGenerate

Global function IlcApply

```
public IlcGoal IlcApply(IlcGoal goal, IlcBranchSelector e)
```

Definition file: ilsolver/search.h
Include file: <ilsolver/ilosolver.h>

This function returns a goal that applies the evaluator `e` to the search tree defined by the goal `goal`. In doing so, it changes the order of evaluation according to `e` of the open nodes of the search tree.

See Also: IlcNodeEvaluator

Global function IloWorstSolutionComparator

`public IloComparator< IloSolution > IloWorstSolutionComparator(IloEnv env)`

Definition file: ilsolver/iimpool.h
Include file: <ilsolver/iim.h>

A solution comparator that prefers lower quality solutions.
This function creates a solution comparator on the environment `env` that prefers lower quality solutions. This comparator makes use of `IloSolution::isWorseThan` in order to make its comparison.

See Also: IloBestSolutionComparator

Global function IloGetClone

```
public X IloGetClone(IloEnvI * env, const X x)
```

Definition file: ilconcert/iloextractable.h

Creates a clone.
This C++ template creates a clone (that is, an exact copy) of an instance of the class X.

Global function IlcNotMember

```
public IlcConstraint IlcNotMember(IlcAny element, IlcAnySetVar var)
public IlcConstraint IlcNotMember(IlcAnyExp element, IlcAnySetVar var)
public IlcConstraint IlcNotMember(IlcIntExp element, IlcIntSetVar var)
public IlcConstraint IlcNotMember(IlcInt element, IlcIntSetVar var)
```

Definition file: ilsolver/ilcset.h
Include file: <ilsolver/ilosolver.h>

This predefined Solver constraint forces a constrained set variable *not* to contain a given element. It does so by creating a constraint that acts on the constrained set variable `var`. When you post `IlcMember`, it prevents the value of `var` from containing `element`. The following cases can arise:

- If `element` belongs to the required set of `var`, then failure occurs.
- If `element` does not belong to the possible set of `var`, then nothing happens.
- In any other case, `element` is removed from the possible set of `var`, and the constraints posted on `var` are activated. If the number of required elements becomes equal to the number of possible elements as a result of this operation, the value of `var` becomes the required set.

See Also: IlcAnyExp, IlcAnySetVar, IlcConstraint, IlcIntExp, IlcIntSetVar, IlcMember, IlcMember

Global function IlcNotMember

```
public IlcConstraint IlcNotMember(const IlcIntExp exp, const IlcIntArray elements)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This predefined Solver constraint forces a constrained integer expression *not* to be an element of the array of integers indicated by `elements`. When you post `IlcNotMember`, it does the opposite of `IlcMember`.

See Also: IlcConstraint, IlcIntExp, IlcMember

Global function IlcGenerate

```
public IlcGoal IlcGenerate(const IlcIntVarArray array, IlcChooseIntIndex
chooseVariable=IlcChooseFirstUnboundInt)
public IlcGoal IlcGenerate(const IlcAnyVarArray array, IlcChooseAnyIndex
chooseVariable=IlcChooseFirstUnboundAny)
public IlcGoal IlcGenerate(const IlcAnyVarArray array, IlcChooseAnyIndex
chooseVariable, IlcAnySelect sel)
public IlcGoal IlcGenerate(const IlcIntVarArray array, IlcChooseIntIndex
chooseVariable, IlcIntSelect select)
public IlcGoal IlcGenerate(const IlcAnySetVarArray array, IlcChooseAnySetIndex
chooseVariable=IlcChooseFirstUnboundAnySet)
public IlcGoal IlcGenerate(const IlcAnySetVarArray array, IlcChooseAnySetIndex
chooseVariable, IlcAnySetSelect select)
public IlcGoal IlcGenerate(const IlcIntSetVarArray array, IlcChooseIntSetIndex
chooseVariable=IlcChooseFirstUnboundIntSet)
public IlcGoal IlcGenerate(const IlcIntSetVarArray array, IlcChooseIntSetIndex
chooseVariable, IlcIntSetSelect select)
public IlcGoal IlcGenerate(const IlcFloatVarArray array, IlcChooseFloatIndex
chooseVariable=IlcChooseFirstUnboundFloat, IlcBool increaseMinFirst=IlcTrue,
IlcFloat prec=0)
public IlcGoal IlcGenerate(IlcBoolVarArray array, IlcChooseBoolIndex
chooseVariable=IlcChooseFirstUnboundBool, IlcBool val=IlcTrue)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

Solver provides an enumeration algorithm with parameters that can be set for choosing the order in which variables are tried during the search for a solution.

This goal binds each constrained variable in its argument `array`; it does so by calling the function `IlcInstantiate` for each of them. The order in which the variables are bound is controlled by the function `chooseVariable`. The argument `select` is passed to each call to `IlcInstantiate`, if that argument is provided.

Implementation

Here's how we could define that goal for `IlcIntVar`.

```
ILCGOAL3(IlcIntGenerate,
         IlcIntVarArray, vars,
         IlcChooseIntIndex,   chooseIndex,
         IlcIntSelectI*, select){
  IlcInt index = chooseIndex(vars);
  if(index == -1) return 0;
  return IlcAnd(IlcIntVarInstantiate(getSolver(),
                                     vars[index],
                                     select),
                this);
}

IlcGoal IlcGenerate(const IlcIntVarArray array,
                    IlcChooseIntIndex chooseIndex){
  return IlcIntGenerate(array.getSolver(),
                        array,
                        chooseIndex,
                        0);
}

IlcGoal IlcGenerate(const IlcIntVarArray array,
                    IlcChooseIntIndex chooseIndex,
                    IlcIntSelect select){
  return IlcIntGenerate(array.getSolver(),
                        array,
                        chooseIndex,
                        select.getImpl());
```

```
}
```

See Also: IlcAnySetVarArray, IlcAnyVarArray, IlcBestGenerate, IlcBestInstantiate, IlcDichotomize, IlcFloatVarArray, IlcGoal, IlcInstantiate, IlcIntSetVarArray, IlcIntVarArray, IlcSolveBounds

Global function IloBestInstantiate

```
public IloGoal IloBestInstantiate(const IloEnv env, const IloNumVar var)
public IloGoal IloBestInstantiate(const IloEnv env, const IloAnyVar var, const
IloAnyValueSelector select)
public IloGoal IloBestInstantiate(const IloEnv env, const IloAnyVar var)
public IloGoal IloBestInstantiate(const IloEnv env, const IloNumVar var, const
IloIntValueSelector select)
public IloGoal IloBestInstantiate(const IloEnv env, const IloIntSetVar var)
public IloGoal IloBestInstantiate(const IloEnv env, const IloAnySetVar var)
public IloGoal IloBestInstantiate(const IloEnv env, const IloIntSetVar var, const
IloIntSetValueSelector select)
public IloGoal IloBestInstantiate(const IloEnv env, const IloAnySetVar var, const
IloAnySetValueSelector select)
```

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal, using the numeric selector indicated by `select`. If the variable `var` has already been bound to a value, then this goal does nothing and succeeds. Otherwise, it sets a choice point and alters the domain of `var` in an attempt to assign a value to `var`. The way the goal alters the domain of `var` varies, depending on the class of `var` and its argument `select`.

When it takes an instance of the class `IloEnv` as a parameter, it returns an instance of `IloGoal` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the goal that it returns as an instance of `IlcGoal` for use during a Solver search.

When the Variable Is Integer or an Enumerated Variable

If its argument `var` has already been bound, `IloBestInstantiate` does nothing and succeeds. Otherwise, it sets a choice point and binds the invoking constrained variable to a value from its domain. If the optional argument `select` is provided, then that value is chosen by `select`; otherwise, the values are tried in ascending order when the values are integers. If a failure occurs, the tried-and-failed value is removed from the domain, and `IloBestInstantiate` succeeds.

When the Variable Is a Set of Variables

If `var` has already been bound, then `IloBestInstantiate` does nothing and succeeds. Otherwise, it sets a choice point, then adds an element of the *possible* set to the *required* set of `var`. The added element is chosen by `select`, if that optional argument is provided; otherwise, the values are tried in ascending order when the values are integers. If failure occurs, the element is removed from the possible set of `var`, and `IloBestInstantiate` succeeds.

Differs from IloInstantiate

This goal differs from the one returned by the function `IloInstantiate`. This goal tries only one value, chosen according to `select`, whereas when a failure occurs, `IloInstantiate` continues to try other values according to `select` until the domain of `var` is exhausted.

> **Note**
>
> Though this function works on numerical variables of type `Float` and type `Int`, it is preferable to use the function `IloDichotomize` with floating-point variables.

See Also: IloGoal, IloInstantiate, IlcBestInstantiate

Global function IloLeLex

```
public IloConstraint IloLeLex(IloEnv, IloIntExprArray, IloIntExprArray, const char
*=0)
```

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

The `IloLeLex` function returns a constraint which maintains two arrays to be lexicographically ordered.

More specifically, `IloLeLex(x, y)` maintains that `x` is less than or equal to `y` in the lexicographical sense of the term. This mean that either both arrays are equal or that there exists `i < size(x)` such that for all `j < i`, `x[j] = y[j]` and `x[i] < y[i]`.

Note that the size of the two arrays must be the same.

See Also: IloGeLex, IlcLeLex, IlcGeLex

Global function IlcBestGenerate

```
public IlcGoal IlcBestGenerate(const IlcIntVarArray, IlcChooseIntIndex
chooseVariable=IlcChooseFirstUnboundInt)
public IlcGoal IlcBestGenerate(const IlcAnyVarArray array, IlcChooseAnyIndex
chooseVariable=IlcChooseFirstUnboundAny)
public IlcGoal IlcBestGenerate(const IlcAnyVarArray array, IlcChooseAnyIndex
chooseVariable, IlcAnySelect select)
public IlcGoal IlcBestGenerate(const IlcIntVarArray, IlcChooseIntIndex
chooseVariable, IlcIntSelect select)
public IlcGoal IlcBestGenerate(const IlcAnySetVarArray array, IlcChooseAnySetIndex
chooseVariable=IlcChooseFirstUnboundAnySet)
public IlcGoal IlcBestGenerate(const IlcAnySetVarArray array, IlcChooseAnySetIndex
chooseVariable, IlcAnySetSelect select)
public IlcGoal IlcBestGenerate(const IlcIntSetVarArray array, IlcChooseIntSetIndex
chooseVariable=IlcChooseFirstUnboundIntSet)
public IlcGoal IlcBestGenerate(const IlcIntSetVarArray array, IlcChooseIntSetIndex
chooseVariable, IlcIntSetSelect select)
public IlcGoal IlcBestGenerate(const IlcFloatVarArray array, IlcChooseFloatIndex
chooseIndex, IlcBool increaseMinFirst=IlcTrue, IlcFloat prec=0)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

Solver provides an enumeration algorithm with parameters that can be set for choosing the order in which variables are tried during the search for a solution. That enumeration algorithm is implemented by the functions `IlcBestGenerate` and `IlcGenerate`.

This goal binds each constrained variable in its argument `array`; it does so by calling the function `IlcBestInstantiate` for each of them. The order in which the variables are bound is controlled by the function `chooseIndex`. The argument `select` is passed to each call to `IlcBestInstantiate`, if that argument is provided.

This goal differs from `IlcGenerate` since it calls `IlcBestInstantiate` which tries only one value.

See Also: IlcAnySelect, IlcAnySetSelect, IlcBestInstantiate, IlcGenerate, IlcGoal, IlcIntSelect, IlcIntSetSelect, IlcSolveBounds

Global function IloLimitSearch

`public IloGoal IloLimitSearch(const IloEnv env, const IloGoal goal, const IloSearchLimit searchLimit)`

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal that limits the exploration of the search tree defined by `goal` in the way directed by `searchLimit`. All nodes explored after that limit has been met are discarded.

When it takes an instance of the class `IloEnv` as a parameter, it returns an instance of `IloGoal` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the goal that it returns as an instance of `IlcGoal` for use during a Solver search.

IBM® ILOG® Solver offers predefined search limits, instances of the class `IloSearchLimit`, such as the return value of the function `IloFailLimit`, `IloOrLimit`, and `IloTimeLimit`.

See Also: IloGoal, IloFailLimit, IloOrLimit, IloSearchLimit, IloTimeLimit, IlcLimitSearch

Global function IlcScalProd

```
public IlcFloatExp IlcScalProd(const IlcFloatVarArray array1, const
IlcFloatVarArray array2)
public IlcIntExp IlcScalProd(const IlcIntVarArray array1, const IlcIntVarArray
array2)
public IlcIntExp IlcScalProd(const IlcIntVarArray array1, const IlcIntArray array2)
public IlcIntExp IlcScalProd(const IlcIntArray array1, const IlcIntVarArray array2)
public IlcInt IlcScalProd(const IlcIntArray array1, const IlcIntArray array2)
public IlcFloatExp IlcScalProd(const IlcFloatVarArray array1, const IlcFloatArray
array2)
public IlcFloatExp IlcScalProd(const IlcFloatArray array1, const IlcFloatVarArray
array2)
public IlcFloat IlcScalProd(const IlcFloatArray array1, const IlcFloatArray array2)
```

Definition file: ilsolver/nlinflt.h
Include file: <ilsolver/ilosolver.h>

This function returns the scalar product of its arguments, that is:

```
(array1[0]*array2[0]) + ... + (array1[size-1]*array2[size-1])
```

When either of its arguments is an array of constrained expressions, then it creates a new constrained expression equal to the scalar product of its arguments.

When both its arguments are arrays of simple unconstrained variables (that is, instances of `IlcIntArray` or `IlcFloatArray`), it merely creates and returns the scalar product.

In any case, the two arrays passed as arguments must have the same number of elements.

The effects of this function are reversible.

See Also: IlcFloatExp, IlcFloatVarArray, IlcIntExp, IlcIntVarArray, IlcSum

Global function IlcBestInstantiate

```
public IlcGoal IlcBestInstantiate(const IlcIntVar var)
public IlcGoal IlcBestInstantiate(const IlcAnyVar var)
public IlcGoal IlcBestInstantiate(const IlcAnyVar var, IlcAnySelect select)
public IlcGoal IlcBestInstantiate(const IlcIntVar var, IlcIntSelect select)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function returns a goal, a primitive in the algorithms that search for solutions. That goal is used to assign a value to a constrained variable.

If its argument `var` has already been bound, `IlcBestInstantiate` does nothing and succeeds. Otherwise, it sets a choice point and binds the invoking constrained variable to a value from its domain. If the optional argument `select` is provided, then that value is chosen by `select`; otherwise, the values are tried in ascending order when the values are integers. If a failure occurs, the tried-and-failed value is removed from the domain, and `IlcBestInstantiate` succeeds.

This function differs from the function `IlcInstantiate` in that respect. `IlcBestInstantiate` tries only one value, whereas when a failure occurs with `IlcInstantiate`, Solver continues trying other values until all the values in the domain of that variable have been tried before it goes on to another variable.

Implementation

This goal can be defined like this for `IlcIntVar`:

```
static
ILCGOAL2(IlcIntBestInstantiate,
         IlcIntVar, var,
         IlcIntSelectI*, select) {
    if (var.isBound()) return 0;
    IlcInt val = (select)? select->select(var) : var.getMin();
    return IlcOr(var == val,
                 var != val);
}
IlcGoal IlcBestInstantiate(IlcIntVar var){
    return IlcIntBestInstantiate(var, 0);
}
IlcGoal IlcBestInstantiate(IlcIntVar var, IlcIntSelect sel){
    return IlcIntBestInstantiate(var, sel.getImpl());
}
```

The code is similar for `IlcAnyVar`.

For more information, see the concept Choice Point.

See Also: IlcBestGenerate, IlcDichotomize, IlcGoal, IlcInstantiate

Global function IlcBestInstantiate

```
public IlcGoal IlcBestInstantiate(const IlcIntSetVar var)
public IlcGoal IlcBestInstantiate(const IlcAnySetVar var)
public IlcGoal IlcBestInstantiate(const IlcAnySetVar var, IlcAnySetSelect select)
public IlcGoal IlcBestInstantiate(const IlcIntSetVar var, IlcIntSetSelect select)
```

Definition file: ilsolver/intset.h
Include file: <ilsolver/ilosolver.h>

This function returns a goal, a primitive in the algorithms that search for solutions. That goal is used to assign a value to a constrained variable. This function differs from the function `IlcInstantiate` (which tries all values in a domain); `IlcBestInstantiate` tries only one possible value. It behaves slightly differently, depending on the class of its arguments.

If `var` has already been bound, then `IlcBestInstantiate` does nothing and succeeds. Otherwise, it sets a choice point, then adds an element of the *possible* set to the *required* set of `var`. The added element is chosen by `select`, if that optional argument is provided; otherwise, the values are tried in ascending order when the values are integers. If failure occurs, the element is removed from the possible set of `var`, and `IlcBestInstantiate` succeeds.

For more information, see the concept Choice Point.

See Also: IlcBestGenerate, IlcGoal, IlcInstantiate

Global function IlcBestInstantiate

`public IlcGoal IlcBestInstantiate(const IlcFloatVar var, IlcBool increaseMinFirst=IlcTrue, IlcFloat prec=0)`

Definition file: ilsolver/nlinflt.h
Include file: <ilsolver/ilosolver.h>

This function returns a goal, a primitive in the algorithms that search for solutions. That goal is used to assign a value to a constrained variable. This function differs from the function `IlcInstantiate` (which tries all values in a domain); `IlcBestInstantiate` tries only one possible value. It behaves slightly differently, depending on the class of its arguments.

If `var` has already been bound, then `IlcBestInstantiate` does nothing and succeeds. Otherwise, it sets a choice point, then replaces the *domain* of `var` by one of its *halves*. If a failure occurs then, the domain is replaced by the other half.

The optional argument `increaseMinFirst` must be a Boolean value, either `IlcTrue` or `IlcFalse`. If it is `IlcTrue`, then the upper half of the domain is tried first; otherwise, the lower half is tried first.

For more information, see the concept Choice Point.

See Also: IlcBestGenerate, IlcDichotomize, IlcGoal, IlcInstantiate

Global function IloChooseMaxRegretMax

`public IlcInt` **`IloChooseMaxRegretMax`**`(const IlcIntVarArray vars)`

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the maximal difference between the maximal possible value and the next maximal possible value from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is based on the principle of *regret*. Regret is the difference between what would have been the best possible decision in a scenario and what was the actual decision.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseIntIndex, IloGenerate

Global function IlcChooseMinSizeFloat

```
public IlcInt IlcChooseMinSizeFloat(const IlcFloatVarArray vars)
```

Definition file: ilsolver/linfloat.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the smallest domain from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcFloatVarArray, IloGenerate

Global function IloDifference

```
public IloIntervalList IloDifference(const IloIntervalList intervals1, const
IloIntervalList intervals2)
```

Definition file: ilconcert/ilointervals.h

Creates and returns the difference between two interval lists.
This operator creates and returns an interval list equal to the difference between the interval list `intervals1` and the interval list `intervals2`. The arguments `intervals1` and `intervals2` must be defined on the same interval. The resulting interval list is defined on the same interval as the arguments. See also `IloIntervalList`.

Global function IloDifference

`public IloNumToAnySetStepFunction **IloDifference**(const IloNumToAnySetStepFunction f1, const IloNumToAnySetStepFunction f2)`

Definition file: ilconcert/ilosetfunc.h

Creates and returns a function equal to the difference between the functions.
This operator creates and returns a function equal to the difference between the functions `f1` and `f2`. The argument functions `f1` and `f2` must be defined on the same interval. The resulting function is defined on the same interval as the arguments. See also: `IloNumToAnySetStepFunction`.

Global function IlcChooseMinRegretMax

`public IlcInt IlcChooseMinRegretMax(const IlcIntVarArray vars)`

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the minimal difference between the maximal possible value and the next maximal possible value from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is based on the principle of *regret*. Regret is the difference between what would have been the best possible decision in a scenario and what was the actual decision.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseIntIndex, IloGenerate

Global function IloSetToValue

```
public IloNHood IloSetToValue(IloEnv env, IloNumVarArray vars, IloNum value, const
char * name=0)
```

Definition file: ilsolver/iimnhood.h
Include file: <ilsolver/ilosolver.h>

This function returns a neighborhood of size `vars.getSize()`, neighbor `i` of which sets `vars[i] = value`. The optional argument `name`, if supplied, becomes the name of the returned neighborhood.

See Also: IloNHood, IloNHoodI

Global function IlcRestartGoal

```
public IlcGoal IlcRestartGoal(IloSolver solver, IlcGoal g, IlcInt failLimit,
IlcFloat factor=1.0)
```

Definition file: ilsolver/ilosolverhandle.h
Include file: <ilsolver/ilosolver.h>

The goal returned by this function performs restart on goal `g`. This goal creates a choice point where the left branch calls `g` with a fail limit of `failLimit` and then calls itself on the right branch with `failLimit = failLimit * factor` (with `factor` remaining unchanged). In other words, it runs `g` with a maximum number of fails equal to `failLimit` and then it runs it again with `failLimit * factor` as the new maximum number of fails and so on. For an example of how to use this goal, see *"Using Impacts during Search"* in the *IBM ILOG Solver User's Manual*.

See Also: IloRestartGoal

Global function IloPower

```
public IloNumExprArg IloPower(const IloNumExprArg base, const IloNumExprArg
exponent)
public IloNumExprArg IloPower(const IloNumExprArg base, IloNum exponent)
public IloNumExprArg IloPower(IloNum base, const IloNumExprArg exponent)
```

Definition file: ilconcert/iloexpression.h

Returns the power of its arguments.
Concert Technology offers predefined functions that return an expression from an algebraic function over expressions. These predefined functions also return a numeric value from an algebraic function over numeric values as well.

`IloPower` returns the result of raising its `base` argument to the power of its `exponent` argument, that is, `base**exponent`. If `base` is a floating-point value or variable, then `exponent` must be greater than or equal to 0 (zero).

What Is Extracted

An instance of `IloCplex` can extract only quadratic terms that are positive semi-definite when they appear in an objective function or in constraints of a model.

An instance of `IloSolver` or an instance of `IloCP` extracts the object returned by `IloPower`.

Global function IloSelectSolutions

`public IloPoolProc **IloSelectSolutions**(IloEnv env, IloSelector< IloSolution, IloSolutionPool > selector, IloBool unique=IloFalse)`

Definition file: ilsolver/iimiloproc.h

Creates a pool processor which selects using a standard Solver selector.
This function creates a pool processor on the environment `env` which selects from its input by asking for successive selections from `selector`. Selectors can be built-in ones such as `IloRandomSelector` or ones created using the `ILOSELECTOR0` or `ILOCTXSELECTOR0` macros (or their variants). (See `ILOSELECTOR0` for `ILOCTXSELECTOR0` documentation.) You can specify if you would like selections to avoid duplicates using the parameter `unique`. Set this flag to `IloTrue` if the pool processor is not to produce duplicates in its output pool.

The following code uses `ILOCTXSELECTOR1` to create a custom selector:

```
ILOCTXSELECTOR1(CustomSelector, IloSolution, IloSolutionPool, in,
                IloSolver, solver, IlcInt, tournamentSize) {
  IloInt size = in.getSize();
  if (size != 0) {
    IlcRandom rnd = solver.getRandom();
    IloSolution candidate = in.getSolution(rnd.getInt(size));
    for (IlcInt tournament = 1; tournament < tournamentSize; tournament++) {
      IloSolution competition = in.getSolution(rnd.getInt(size));
      if (competition.isBetterThan(candidate))
        candidate = competition;
    }
    select(candidate);
  }
}
```

If `selector` has been created from `ILOCTXSELECTOR1`, `IloSelectSolutions` will ensure that the context passed will be the instance of the `IloSolver` on which it is executing.

See Also: IloSelector, ILOSELECTOR0

Global function IlcSetOf

`public IlcIntSetVar IlcSetOf(IlcIndex & i, IlcConstraint ct)`

Definition file: ilsolver/intset.h
Include file: <ilsolver/ilosolver.h>

You can use the function `IlcSetOf` to reason about a set of constrained variables that all depend on a given generic constraint.

This function creates a constrained integer set variable, `setvar`, equal to the set of integer values `j` such that the generic constraint denoted by `ct` is true for `j`. At all times, all integer values `j` such that `ct` is true for `j` belong to the *required* set of `setvar`; and all integer values `k` such that `ct` is not false for `k` belong to the *possible* set of `setvar`.

The generic constraint `ct` must have been created with generic variables *stemming from* the *index*`i`; otherwise, Solver will throw an exception (an instance of `IloSolver::SolverErrorException`).

All generic variables of the constraint `ct` must represent arrays of constrained expressions of the same size; otherwise, Solver will throw an exception (an instance of `IloSolver::SolverErrorException`).

Generic Constraints

A *generic constraint* is a constraint shared by an array of variables. For example, `IlcAllDiff` is a generic constraint that insures that all the elements of a given array are different from one another. Solver provides generic constraints to save memory since, if you use them, you can avoid allocating one object per variable.

You create a generic constraint simply by stating the constraint over *generic variables*. Each generic variable stands for all the elements of an array of constrained variables.

In that sense, generic variables are only syntactic objects provided by Solver to support generic constraints, and they can be used only for creating generic constraints. To create a generic variable, you use the operator `[]`.The argument passed to that operator is known as the *index* for that generic variable; we say that the generic variable *stems from* that index.

Example

Let's assume we have two arrays, `A` and `B`, of constrained integer variables (that is, instances of `IlcIntExp` or its subclasses). We want to know the set of elements in those arrays such that `A[i] > B[i]`. All we have to do is this:

```
IlcIndex i(solver);
IlcIntSetVar s = IlcSetOf(i, A[i] > B[i]);
```

That constrained set variable `s` can, in turn, be constrained by other constraints. For example, we can set its cardinality (the number of elements in it) to 0 (zero). Doing that amounts to saying that no such `i` exists such that `A[i] > B[i]`. In other words, for all `i`, `A[i] <= B[i]`.

There is a “shortcut” for constraining the cardinality of the set for which a given constraint is true. That shortcut is implemented by the function `IlcCard`.

See Also: IlcCard, IlcDistribute, IlcIndex

Global function IloSelectSearch

```
public IloGoal IloSelectSearch(const IloEnv env, const IloGoal g, const
IloSearchSelector s)
```

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function returns a goal that applies the selector `s` to the search tree defined by the goal `g`. As the goal handler explores the search tree, it gives successful leaves to the selector. When the tree has been fully explored, the selector is called, and it re-activates the selected nodes.

When it takes an instance of the class `IloEnv` as a parameter, it returns an instance of `IloGoal` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the goal that it returns as an instance of `IlcGoal` for use during a Solver search.

See Also: IlcSearchSelector, IloFirstSolution, IloSearchSelector, IlcSelectSearch

Global function IloMaximize

```
public IloObjective IloMaximize(const IloEnv env, IloNum constant=0.0, const char * name=0)
public IloObjective IloMaximize(const IloEnv env, const IloNumExprArg expr, const char * name=0)
```

Definition file: ilconcert/ilolinear.h

Defines a maximization objective.
This function defines a maximization objective in a model. In other words, it simply offers a convenient way to create an instance of `IloObjective` with its sense defined as `Maximize`. However, an instance of `IloObjective` created by `IloMaximize` may not necessarily maintain its sense throughout the lifetime of the instance. The optional argument `name` is set to `0` by default.

You may define more than one objective in a model. However, algorithms conventionally take into account only one objective at a time.

Global function IloLog

```
public IloNumExprArg IloLog(const IloNumExprArg arg)
public IloNum IloLog(IloNum val)
```

Definition file: ilconcert/iloexpression.h

Returns the natural logarithm of its argument.
Concert Technology offers predefined functions that return an expression from an algebraic function on expressions. These predefined functions also return a numeric value from an algebraic function on numeric values as well.

`IloLog` returns the natural logarithm of its argument. In order to conform to IEEE 754 standards for floating-point arithmetic, you should use this function in your Concert Technology applications, rather than the standard C++ `log`.

Global function IlcReductionVarEvaluator

```
public IloEvaluator< IlcIntVar > IlcReductionVarEvaluator(IloEnv env)
public IloEvaluator< IlcIntVar > IlcReductionVarEvaluator(IloSolver solver)
```

Definition file: ilsolver/custgoal.h
Include file: <ilsolver/ilosolver.h>

This function returns an instance of the evaluator `IloEvaluator<IlcIntVar>`. The evaluation returns the value returned by `solver.getReduction(x)`, where `x` is the evaluated variable.

Global function IlcMember

```
public IlcConstraint IlcMember(const IlcIntExp exp, const IlcIntArray elements)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This predefined Solver constraint forces the domain of `exp` to be a subset of or equal to the array `elements`. When you post `IlcMember`, it constrains each value of `exp` to belong to `elements`. In other words, any value of `exp` that is not in the array `elements` will be removed from the domain of `exp`.

See Also: IlcConstraint, IlcIntArray, IlcIntExp, IlcMember, IlcNotMember

Global function IlcMember

```
public IlcConstraint IlcMember(IlcAny element, IlcAnySetVar setVar)
public IlcConstraint IlcMember(IlcAnyExp element, IlcAnySetVar setVar)
public IlcConstraint IlcMember(IlcIntExp element, IlcIntSetVar setVar)
public IlcConstraint IlcMember(IlcInt element, IlcIntSetVar setVar)
```

Definition file: ilsolver/ilcset.h
Include file: <ilsolver/ilosolver.h>

This predefined Solver constraint forces a constrained set variable to contain a given element. It does so by creating a constraint that acts on the constrained set variable `setVar`. When you post `IlcMember`, it constrains the value of `setVar` to contain `element`. The following cases can arise:

- If `element` belongs to the required set of `setVar`, then nothing happens.
- If `element` does not belong to the possible set of `setVar`, then failure occurs.
- In any other case, `element` is added to the required set of `setVar`, and the constraints posted on `setVar` are activated. If the number of required elements becomes equal to the number of possible elements as a result of this operation, the value of `setVar` becomes the required set.

See Also: IlcAnyExp, IlcAnySetVar, IlcConstraint, IlcIntExp, IlcIntSetVar, IlcMember, IlcNotMember

Global function IloDisableNANDetection

```
public void IloDisableNANDetection()
```

Definition file: ilconcert/ilosys.h

Disables NaN (Not a number) detection.
This function turns off NaN (Not a number) detection.

Global function operator<<

```
public ostream & operator<<(ostream & out, IloAlgorithm::Status st)
public ostream & operator<<(ostream & out, const IloArray< X > & a)
public ostream & operator<<(ostream & out, const IloNumExpr & ext)
public ostream & operator<<(ostream & os, const IloRandom & r)
public ostream & operator<<(ostream & stream, const IloSolution & solution)
public ostream & operator<<(ostream & stream, const IloSolutionManip & fragment)
public ostream & operator<<(ostream & o, const IloException & e)
```

Definition file: ilconcert/iloalg.h

Overloaded C++ operator.
This overloaded C++ operator directs output to an output stream.

Global function operator<<

```
public ostream & operator<<(ostream & str, const IlcAnyArray & exp)
public ostream & operator<<(ostream & str, const IlcAnySet & exp)
public ostream & operator<<(ostream & str, const IlcAnyExp exp)
public ostream & operator<<(ostream & str, const IlcConstAnyArray & exp)
public ostream & operator<<(ostream & str, const IlcAnyVarArray & exp)
public ostream & operator<<(ostream & str, IlcExpArrayI & array)
public ostream & operator<<(ostream & str, const IlcGoal & f)
public ostream & operator<<(ostream & str, const IlcGoalI & f)
public ostream & operator<<(ostream & str, const IlcDemon & f)
public ostream & operator<<(ostream & str, const IlcDemonI & f)
public ostream & operator<<(ostream & str, const IlcConstraint & f)
public ostream & operator<<(ostream & str, const IlcConstraintArray & exp)
public ostream & operator<<(ostream & str, const IlcFloatSet & exp)
public ostream & operator<<(ostream & str, const IlcFloatExp & exp)
public ostream & operator<<(ostream & str, const IlcFloatVarArray & exp)
public ostream & operator<<(ostream & str, const IlcFloatArray & exp)
public ostream & operator<<(ostream & str, const IlcConstFloatArray & exp)
public ostream & operator<<(ostream & str, const IlcFloatExpI & var)
public ostream & operator<<(ostream & stream, IloNHood nhood)
public ostream & operator<<(ostream & str, const IlcAnySetVar & exp)
public ostream & operator<<(ostream & str, const IlcAnySetVarArray & exp)
public ostream & operator<<(ostream & str, const IlcAnySetArray & exp)
public ostream & operator<<(ostream & str, const IlcIntSet & exp)
public ostream & operator<<(ostream & str, const IlcIntExp & exp)
public ostream & operator<<(ostream & str, const IlcIntArray & exp)
public ostream & operator<<(ostream & str, const IlcConstIntArray & exp)
public ostream & operator<<(ostream & str, const IlcIntVarArray & exp)
public ostream & operator<<(ostream & str, const IlcIntExpI & var)
public ostream & operator<<(ostream & str, const IlcIntSetVar & exp)
public ostream & operator<<(ostream & str, const IlcIntSetVarArray & exp)
public ostream & operator<<(ostream & str, const IlcIntSetArray & exp)
public ostream & operator<<(ostream & str, const IlcExprI & var)
```

Definition file: ilsolver/anyexp.h
Include file: <ilsolver/ilosolver.h>

This operator directs output to an output stream, usually standard output.

See Also: IlcAnyArray, IlcAnyExp, IlcAnySet, IlcAnySetVar, IlcAnySetVarArray, IlcAnyVarArray, IlcConstAnyArray, IlcConstFloatArray, IlcConstIntArray, IlcConstraintArray, IlcFloatArray, IlcFloatExp, IlcFloatVarArray, IlcGoal, IlcGoalI, IlcIntArray, IlcIntExp, IlcIntSet, IlcIntSetArray, IlcIntSetVar, IlcIntSetVarArray, IlcIntVarArray, IloNHood

Global function operator<<

```
public ostream & operator<<(ostream & out, const IloCsvLine & line)
```

Definition file: ilconcert/ilocsvreader.h

Overloaded operator for csv output.
This operator has been overloaded to treat an `IloCsvLine` object appropriately as output. It directs its output to an output stream (normally, standard output) and displays information about its second argument `line`.

Global function operator<<

```
public IloEvaluator< IloObjectIn > operator<<(IloEvaluator< IloObject > e,
IloTranslator< IloObject, IloObjectIn > t)
```

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

This operator creates a composite `IloEvaluator<IloObjectIn>` instance. This evaluator first translates an instance of class `IloObjectIn` into an instance of class `IloObject` using the translator, and then uses the evaluator given in the left side to evaluate this instance.

For more information, see Selectors.

Global function operator<<

```
public ostream & operator<<(ostream & out, const IloExtractable & ext)
```

Definition file: ilconcert/iloextractable.h

Overloaded C++ operator.
This overloaded C++ operator directs output to an output stream.

Global function operator<<

```
public IloPredicate< IloObjectIn > operator<<(IloPredicate< IloObject > e,
IloTranslator< IloObject, IloObjectIn > t)
```

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

Creates translated predicate.
This predicate first translates an instance of class `IloObjectIn` into an instance of class `IloObject` using the translator `t`, and then uses the predicate `e` given at the left side to test this instance.

For more information, see Selectors.

Global function operator-

```
public IlcIntExp operator-(const IlcIntExp exp)
public IlcFloatExp operator-(const IlcFloatExp exp)
public IlcIntToIntStepFunction operator-(const IlcIntToIntStepFunction & f1)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This arithmetic operator returns the opposite of its argument. It has been overloaded to handle constrained expressions appropriately. The domain of the resulting expression is computed from the domain of the original expression as you would expect.

See Also: IlcIntExp, IlcFloatExp, IlcIntToIntStepFunction

Global function operator-

```
public IloEvaluator< IloObject > operator-(IloEvaluator< IloObject > left,
IloEvaluator< IloObject > right)
public IloEvaluator< IloObject > operator-(IloEvaluator< IloObject > left, IloNum
c)
public IloEvaluator< IloObject > operator-(IloNum c, IloEvaluator< IloObject >
right)
public IloEvaluator< IloObject > operator-(IloEvaluator< IloObject > eval)
```

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

These operators create a composite `IloEvaluator<IloObject>` instance. The semantics of the new evaluator are the subtraction of the values of the component evaluators. The first function combines two evaluators, subtracting the value returned by the first by the value returned by the second. The next two functions combine an `IloNum` value with an evaluator. In the first of these functions, the `IloNum` value is subtracted from the value returned by the evaluator, while in the second of these functions those roles are reversed. The last function returns the opposite of an evaluator.

For more information, see Selectors.

Global function operator-

```
public IloNumToNumStepFunction operator-(const IloNumToNumStepFunction f1, const IloNumToNumStepFunction f2)
```

Definition file: ilconcert/ilonumfunc.h

Creates and returns a function equal to the difference between its argument functions.
This operator creates and returns a function equal to the difference between functions `f1` and `f2`. The argument functions `f1` and `f2` must be defined on the same interval. The resulting function is defined on the same interval as the arguments. See also: `IloNumToNumStepFunction`.

Global function operator-

```
public IlcFloatExp operator-(const IlcFloatExp exp1, IlcFloat exp2)
public IlcIntExp operator-(const IlcIntExp exp1, const IlcIntExp exp2)
public IlcIntExp operator-(const IlcIntExp exp1, IlcInt exp2)
public IlcIntExp operator-(IlcInt exp1, const IlcIntExp exp2)
public IlcFloatExp operator-(IlcFloat exp1, const IlcFloatExp exp2)
public IlcFloatExp operator-(const IlcFloatExp exp1, const IlcFloatExp exp2)
public IlcIntToIntStepFunction operator-(const IlcIntToIntStepFunction & f1, const
IlcIntToIntStepFunction & f2)
```

Definition file: ilsolver/linfloat.h
Include file: <ilsolver/ilosolver.h>

This arithmetic operator subtracts its second argument from its first. It has been overloaded to handle constrained expressions appropriately. The domain of the resulting expression is computed from the domains of the combined expressions as you would expect.

See Also: IlcFloatExp, IlcIntExp, IlcIntToIntStepFunction

Global function operator-

```
public IloNumExprArg operator-(const IloNumExprArg x, const IloNumExprArg y)
public IloNumExprArg operator-(const IloNumExprArg x, IloNum y)
public IloNumExprArg operator-(IloNum x, const IloNumExprArg y)
public IloIntExprArg operator-(const IloIntExprArg x, const IloIntExprArg y)
public IloIntExprArg operator-(const IloIntExprArg x, IloInt y)
public IloIntExprArg operator-(IloInt x, const IloIntExprArg y)
```

Definition file: ilconcert/iloexpression.h

Returns an expression equal to the difference of its arguments.
This overloaded C++ operator returns an expression equal to the difference of its arguments. Its arguments may be numeric values, numeric variables, or other expressions.

Global function IloComposePareto

```
public IloParetoComparator< IloObject > IloComposePareto(IloComparator< IloObject >
a, IloComparator< IloObject > b)
public IloParetoComparator< IloObject > IloComposePareto(IloComparator< IloObject >
a, IloComparator< IloObject > b, IloComparator< IloObject > c)
public IloParetoComparator< IloObject > IloComposePareto(IloComparator< IloObject >
a, IloComparator< IloObject > b, IloComparator< IloObject > c, IloComparator<
IloObject > d)
public IloParetoComparator< IloObject > IloComposePareto(IloComparator< IloObject >
a, IloComparator< IloObject > b, IloComparator< IloObject > c, IloComparator<
IloObject > d, IloComparator< IloObject > e)
```

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

Initializes a Pareto composite comparator from existing comparators.
This function creates a Pareto composite comparator from existing comparators.

For more information, see Selectors.

See Also: IloParetoComparator

Global function IloSelectProcessor

```
public IloPoolProc IloSelectProcessor(IloEnv env, IloPoolProcArray procs,
IloSelector< IloPoolProc, IloPoolProcArray > selector, const char * name=0)
```

Definition file: ilsolver/iimiloproc.h

A pool processor which is a selection from a set of others.
This function creates a pool processor which will act as one of a set of given processors each time it is asked to produce solutions. The choice of processor is made by a selector object.

The processor generated by `IloSelectProcessor` works as follows. First, the pool processor to the processor's right (as defined by operator `>>`) asks the processor for *n* solutions. The processor then performs the following instructions repetitively until at least *n* solutions have been produced (placed on the output pool of the processor).

- The processor selects a processor *p* from `procs` by calling `selector.select(p, procs)`.
- The processor asks *p* to produce its natural number of solutions.
- The solutions produced by *p* are placed on the output pool of the processor.

Global function IlcChooseMinSizeInt

`public IlcInt IlcChooseMinSizeInt(const IlcIntVarArray vars)`

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the smallest domain from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseIntIndex, IloGenerate

Global function IlcDistribute

```
public IlcConstraint IlcDistribute(IlcIntVarArray cards, IlcAnyArray values,
IlcAnyVarArray vars)
public IlcConstraint IlcDistribute(IlcIntVarArray cards, IlcAnyArray values,
IlcAnyVarArray vars, IlcFilterLevel level)
public IlcConstraint IlcDistribute(IlcIntVarArray cards, IlcIntArray values,
IlcIntVarArray vars, IlcFilterLevel level)
public IlcConstraint IlcDistribute(IlcIntVarArray cards, IlcIntVarArray vars,
IlcFilterLevel level)
public IlcConstraint IlcDistribute(IlcIntVarArray cards, IlcIntArray values,
IlcIntVarArray vars)
public IlcConstraint IlcDistribute(IlcIntVarArray cards, IlcIntVarArray vars)
```

Definition file: ilsolver/ilcany.h
Include file: <ilsolver/ilosolver.h>

You can use the function `IlcDistribute` to count the number of occurrences of several values among the constrained variables in an array of constrained variables. (See the functions `IlcCard` and `IlcSetOf` for other counting constraints.)

This function creates and returns a constraint. That constraint has no effect until you post it. When this constraint is posted, then the constrained variables in the array `cards` are equal to the number of occurrences in the array `vars` of the values in the array `values`. More precisely, for each `i`, `cards[i]` is equal to the number of occurrences of `values[i]` in the array `vars`. After propagation of this constraint, the minimum of `cards[i]` is at least equal to the number of variables contained in `vars` bound to the value at `values[i]`; and the maximum of `cards[i]` is at most equal to the number of variables contained in `vars` that contain the value at `values[i]` in their domain.

The arrays `cards` and `values` must be the same length; otherwise, Solver will throw an exception (an instance of `IloSolver::SolverErrorException`).

When this function has only two arguments (that is, there is no `values` parameter), then the array of values that are being counted must be an array of consecutive integers starting with 0 (zero). In that case, for each `i`, `cards[i]` is equal to the number of occurrences of `i` in the array `vars`. After propagation of this constraint, the minimum of `cards[i]` is at least equal to the number of variables contained in `vars` bound to the value `i`; and the maximum of `cards[i]` is at most equal to the number of variables contained in `vars` that contain `i` in their domain.

If you do not explicitly state a filter level, then Solver will use the default filter level for this constraint. The optional argument `level` can take either of two values: `IlcBasic` or `IlcExtended`. Its lowest value is `IlcBasic`. The amount of domain reduction during propagation depends on that value. See `IlcFilterLevel` for an explanation of filter levels and their effect on constraint propagation.

`IlcBasic` makes the statement

```
s.add(IlcDistribute(cards, values, vars));
```

more efficient and causes more domain reductions than the following code:

```
IlcIndex j;
IlcInt size = cards.getSize();
for (IlcInt i = 0; i < size; i++)
     s.add(cards[i] == IlcCard(j, vars[j] == values[i])
);
```

`IlcExtended` causes more domain reduction than `IlcBasic`; it also takes longer to run.

Adding These Constraints

You may add these constraints only during a Solver search; that is, inside a goal (an instance of `IlcGoal`) or inside a constraint (an instance of `IlcConstraint`). If you are looking for similar functionality in a constraint to add to a *model*, see `IloDistribute` documented in the *Concert Technology Reference Manual*.

Programming Hint

This statement:

```
s.add(IlcDistribute(cards, values, vars));
```

is more efficient than but equivalent to the following code:

```
assert(cards.getSize() == values.getSize());
IlcInt i;
IlcIndex j;
IlcInt size = cards.getSize();
for(i=0; i<size; i++)
    s.add(cards[i] == IlcCard(j, vars[j] == values[i]));
```

Programming Hint

This statement:

```
s.add(IlcDistribute(cards, vars));
```

is more efficient than but equivalent to the following code:

```
IlcInt size = cards.getSize();
IlcIntArray values(size);
for(IlcInt i = 0; i < size; i++)
    values[i] = i;
s.add(IlcDistribute(cards, values, vars));
```

See Also: IlcAbstraction, IlcCard, IlcConstraint, IlcFilterLevel, IlcIndex, IlcSequence, IlcSetOf

Global function IlcLeLex

`public IlcConstraint IlcLeLex(IlcIntVarArray x, IlcIntVarArray y)`

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

The `IlcLeLex` function returns a constraint which maintains two arrays to be lexicographically ordered.

More specifically, `IlcLeLex(x, y)` maintains that `x` is less than or equal to `y` in the lexicographical sense of the term. This mean that either both arrays are equal or that there exists `i < size(x)` such that for all `j < i`, `x[j] = y[j]` and `x[i] < y[i]`.

Note that the size of the two arrays must be the same.

See Also: IloGeLex, IloLeLex, IlcGeLex

Global function IlcArcSin

```
public IlcFloatExp IlcArcSin(const IlcFloatExp x)
public IlcFloat IlcArcSin(IlcFloat x)
```

Definition file: ilsolver/nlinflt.h
Include file: <ilsolver/ilosolver.h>

When its argument is a constrained floating-point expression, this function creates a constrained floating-point expression (that is, an instance of `IlcFloatExp` or one of its subclasses) which is equal to the arc sine (in the range -Pi/2 to Pi/2) of its argument `x` expressed in radians. The effects of this function are reversible.

When its argument is an unconstrained numeric value (that is, a value of type `IlcFloat`), this function returns the arc sine of its argument.

If you want to manipulate constrained floating-point expressions in degrees, we strongly recommend that you call the trigonometric functions on variables expressed in radians and then convert the results to degrees (rather than declaring the constrained floating-point expressions in degrees and then converting them to radians to call the trigonometric functions).

The reason for that advice is that the method we recommend gives more accurate results in the context of the usual floating-point pitfalls.

See Also: IlcArcCos, IlcArcTan, IlcCos, IlcDegToRad, IlcFloatExp, IlcHalfPi, IlcPi, IlcQuarterPi, IlcRadToDeg, IlcSin, IlcTan, IlcThreeHalfPi, IlcTwoPi

Global function operator new

`public void * operator new(size_t sz, const IloEnv & env)`

Definition file: ilconcert/iloenv.h

Overloaded C++ `new` operator.
IBM® ILOG® Concert Technology offers this overloaded C++ `new` operator. This operator is overloaded to allocate data on internal data structures associated with an invoking environment (an instance of `IloEnv`). The memory used by objects allocated with this overloaded operator is automatically reclaimed when you call the member function `IloEnv::end`. As a developer, you must *not* delete objects allocated with this operator because of this automatic freeing of memory.

In other words, you must *not* use the `delete` operator for objects allocated with this overloaded `new` operator.

The use of this overloaded `new` operator is not obligatory in Concert Technology applications. You will see examples of its use in the user's manuals that accompany the IBM® ILOG® optimization products.

Global function operator new

```
public void * operator new(size_t s, IlcAllocationStack * heap)
```

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>

Solver provides an overloaded `new` operator. This operator is overloaded to allocate data on the heap associated with the invoking solver (an instance of `IloSolver`). The memory used by objects allocated with this operator is automatically reclaimed in these situations:

- whenever Solver backtracks to a choice point set previously;
- when a model (an instance of `IloModel`) or other extractable objects (instances of `IloExtractable` or its subclasses) is re-extracted for a solver (an instance of `IloSolver`);
- when the member function `end` is called for the invoking solver.

As a developer, you must *not* delete objects allocated with this operator because of this automatic freeing of memory.

In other words, the `delete` operator must *not* be used for objects allocated on the heap associated with a solver.

The use of this overloaded `new` operator is *not* obligatory. In fact, the use of the solver heap is not mandatory. You determine whether Solver uses the overloaded `new` operator or the conventional C++ `new` operator when you call the member function `useHeap`. In particular, you can allocate instances of Solver classes using the standard `new` operator or even a special purpose allocator. However, some Solver objects contain other objects. For example, Solver variables contain other objects (finite sets) that represent their domains. These subobjects are allocated onto the solver allocation heap. Likewise, constraints are allocated onto the solver allocation heap. Thus, they must not be deleted. Solver manages the corresponding memory transparently.

To allocate an array of `size` objects of type `T` on the solver allocation heap, you simply write this:

```
T* array = new (s.getHeap()) T [size];
```

When you do not want to use the solver allocation heap, you write this:

```
T* array = new T [size];
```

When you allocate an array in the way we recommend, it will automatically be de-allocated in either of two situations: if backtracking occurs to a choice point set before this allocation; if the member function `end` is called for the invoking solver.

See Also: IlcMemoryManagerI

Global function IloMax

```
public IloEvaluator< IloObject > IloMax(IloEvaluator< IloObject > left,
IloEvaluator< IloObject > right)
public IloEvaluator< IloObject > IloMax(IloEvaluator< IloObject > left, IloNum c)
public IloEvaluator< IloObject > IloMax(IloNum c, IloEvaluator< IloObject > right)
```

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

These functions create a composite `IloEvaluator<IloObject>` instance. These evaluators return the greatest value of the float values returned by the two evaluators given as argument, or the least value between the float value and the evaluator given as argument.

For more information, see Selectors.

Global function IloMax

```
public IloNum IloMax(const IloNumArray vals)
public IloNum IloMax(IloNum val1, IloNum val2)
public IloInt IloMax(const IloIntArray vals)
public IloNumExprArg IloMax(const IloNumExprArray exprs)
public IloIntExprArg IloMax(const IloIntExprArray exprs)
public IloNumExprArg IloMax(const IloNumExprArg x, const IloNumExprArg y)
public IloNumExprArg IloMax(const IloNumExprArg x, IloNum y)
public IloNumExprArg IloMax(IloNum x, const IloNumExprArg y)
public IloIntExprArg IloMax(const IloIntExprArg x, const IloIntExprArg y)
public IloIntExprArg IloMax(const IloIntExprArg x, IloInt y)
public IloNumExprArg IloMax(const IloIntExprArg x, IloNum y)
public IloIntExprArg IloMax(const IloIntExprArg x, int y)
public IloIntExprArg IloMax(IloInt x, const IloIntExprArg y)
public IloNumExprArg IloMax(IloNum x, const IloIntExprArg y)
public IloIntExprArg IloMax(int x, const IloIntExprArg y)
```

Definition file: ilconcert/iloexpression.h

Returns a numeric value representing the max of numeric values.
These functions compare their arguments and return the greatest value.

Global function IloMax

```
public IloNumToNumStepFunction IloMax(const IloNumToNumStepFunction f1, const
IloNumToNumStepFunction f2)
```

Definition file: ilconcert/ilonumfunc.h

Creates and returns a function equal to the maximal value of its argument functions.
This operator creates and returns a function equal to the maximal value of the functions `f1` and `f2`. That is, for all points `x` in the definition interval, the resulting function is equal to the `max(f1(x), f2(x))`. The argument functions `f1` and `f2` must be defined on the same interval. The resulting function is defined on the same interval as the arguments. See also: `IloNumToNumStepFunction`.

Global function IloTableConstraint

```
public IloConstraint IloTableConstraint(const IloEnv env, const IloIntVarArray
vars, const IloIntTupleSet set, IloBool compatible)
public IloConstraint IloTableConstraint(const IloEnv env, const IloIntVar var1,
const IloIntVar var2, const IloIntTupleSet set, IloBool compatible)
public IloConstraint IloTableConstraint(const IloEnv env, const IloIntVar var1,
const IloIntVar var2, const IloIntVar var3, const IloIntTupleSet set, IloBool
compatible)
public IloConstraint IloTableConstraint(const IloEnv env, const IloIntVarArray
vars, const IloIntTernaryPredicate pred)
public IloConstraint IloTableConstraint(const IloEnv env, const IloIntVar var1,
const IloIntVar var2, const IloIntVar var3, const IloIntTernaryPredicate pred)
public IloConstraint IloTableConstraint(const IloEnv env, const IloIntVarArray
vars, const IloIntBinaryPredicate pred)
public IloConstraint IloTableConstraint(const IloEnv env, const IloIntVar var1,
const IloIntVar var2, const IloIntBinaryPredicate pred)
public IloConstraint IloTableConstraint(const IloEnv env, const IloIntVar y, const
IloIntArray a, const IloNumVar x)
public IloConstraint IloTableConstraint(const IloEnv env, const IloAnyVarArray
vars, const IloAnyTupleSet set, IloBool compatible)
public IloConstraint IloTableConstraint(const IloEnv env, const IloAnyVarArray
vars, const IloAnyTernaryPredicate pred)
public IloConstraint IloTableConstraint(const IloEnv env, const IloAnyVarArray
vars, const IloAnyBinaryPredicate pred)
public IloConstraint IloTableConstraint(const IloEnv env, const IloAnyVar y, const
IloAnyArray a, const IloNumVar x)
```

Definition file: ilconcert/ilotupleset.h

For IBM® ILOG® Solver: defines simple constraints that are not predefined.
This function can be used to define simple constraints that are not predefined. It creates and returns a constraint for use in an IBM ILOG Concert Technology model. That constraint is defined for *all* the constrained variables in the array `vars` or for the single constrained variable `y`.

This kind of constraint is sometimes known as an *element* constraint.

The semantics of that generic constraint can be specified in either one of several ways:

- by a predicate; in that case, the argument `pred` specifies that predicate;
- by the values that satisfy the constraint; in that case, the argument `set` specifies the combinations of values that satisfy the constraint, and the argument `compatible` must be `IloTrue`;
- by the values that do *not* satisfy the constraint; in that case, the argument `set` specifies the unsatisfactory combinations of values, and the argument `compatible` must be `IloFalse`;
- by making the constrained variable `y` equal to the element of the array `a` at the index specified by `x`. In other words, `y=a[x];`

The order of the constrained variables in the array `vars` is important because the same order is respected in the predicate `pred` or the `set`. That is, `IloTableConstraint` passes an array of values to the member function `isTrue` for a predicate or to the member function `isIn` for a set, where the first such value is a value of `vars[0]`, the second is a value of `vars[1]`, and in general, the *i-th* value is a value of the constrained variable `vars[i]`.

To avoid exceptions, you must observe the following conditions:

- If the function is called with a predicate `pred` as an argument, the size of the array of constrained variables must be three.
- The size of `vars` is must be the same as the size of the `set`.

Global function IlcPathLength

`public IlcConstraint IlcPathLength(IlcIntVarArray next, IlcFloatVarArray lengths, IlcPathTransit transit, IlcInt maxNbPaths, IlcWhenEvent event=IlcWhenValue)`

Definition file: ilsolver/ilcpath.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a *path* constraint. Like other Solver constraints, this one must be posted in order to be taken into account.

What IlcPathLength Does Not Do

The constraint that this function returns does not determine whether there is a path between nodes in a graph; rather, it constrains accumulations (such as flow) along a path. The filtering algorithm associated with this constraint works on the accumulation variables in the array `lengths`.

If you are looking for a Hamiltonian path, for example, (that is, one in which each node is visited exactly once), consider using instead the constraint `IlcAllDiff` on the variables in the array `next`.

What IlcPathLength Does

If we are given

- a set of n nodes, known as N,
- a maximum number of paths among those nodes, `maxNbPaths`,
- a set of `maxNbPaths` nodes, known as S, for starting nodes,
- a set of `maxNbPaths` nodes, known as E, for ending nodes,

then a path constraint insures that there exist at most `maxNbPaths` paths starting from a node in S, visiting nodes in N, and ending at a node in E. Furthermore, each node will be *visited* only once, has only one predecessor and only one successor, and each node *belongs* to a path that starts from a node in S and ends at a node in E.

In particular, in the function `IlcPathLength`, in the arrays `next` and `lengths`,

- the indices in `[0, n-1]` correspond to the nodes of N,
- the indices in `[n, n+maxNbPaths-1]` correspond to the nodes of E,
- and the indices in `[n+maxNbPaths, n+2*maxNbPaths-1]` correspond to the nodes of S.

In other words, the size of `next` and `lengths` is `n+2*maxNbPaths`.

`next[i]` is the node following node `i` on the current path. `lengths[i]` is the accumulated cost from the beginning of the path to node `i`. The argument `transit` indicates the *transition function*.

When you post this constraint, it insures that for all indices `i` in the range `[0, n-1]` or in `[n+maxNbPaths, n+2*maxNbPaths-1]`, if `next[i]==j` and j is in `[0, n+maxNbPaths-1]`, then `lengths[i] + transit.transit(i,j) <= lengths[j]`.

When `i` is in the range `[n, n+maxNbPaths-1]`, `next[i]` has no meaning because the nodes in E do not have successors, of course. In this case, the constraint deals with them by setting `next[i]` to `i+maxNbPaths` (that is, nodes of S).

The argument `event` can take one of two values: `IlcWhenValue` or `IlcWhenDomain` (values of the enumeration `IlcWhenEvent`). The behavior of the constraint depends on the chosen value.

- `IlcWhenValue` means that the constraint is associated with the value propagation event of each one of the constrained variables of `next`.
- `IlcWhenDomain` means that the constraint is associated with the domain propagation event of each one of the constrained variables of next. This choice causes more domain reduction than the preceding choice and in consequence takes longer to run.

If you set the number of paths to zero (`maxNbPaths = 0`), then you can have as many paths as needed, in case you do not know the number in advance. In such a case, a node i is at the end of a path if `next[i]>=next.getSize()`. Solver recognizes the starting node of such a path by the fact that there is no node `j` such that `next[j]==i`.

See Also: IlcConstraint, IlcInverse, IlcPathTransit, IlcPathTransitEvalI, IlcPathTransitFunction, IlcPathTransitI

Global function operator==

```
public IloConstraint operator==(const IloAnyVar var1, const IloAnyVar var2)
public IloConstraint operator==(const IloAnyVar var1, IloAny val)
public IloConstraint operator==(IloAny val, const IloAnyVar var1)
public IloConstraint operator==(const IloAnySetVar var1, const IloAnySetVar var2)
public IloConstraint operator==(const IloAnySetVar var1, const IloAnySet set)
public IloConstraint operator==(const IloAnySet set, const IloAnySetVar var1)
public IloConstraint operator==(const IloIntSetVar var1, const IloIntSetVar var2)
public IloConstraint operator==(const IloIntSetVar var1, const IloIntSet set)
public IloConstraint operator==(const IloIntSet set, const IloIntSetVar var)
```

Definition file: ilconcert/iloany.h

This overloaded C++ operator constrains its two arguments to be equal. In order to be taken into account, this constraint must be added to a model and extracted for an algorithm.

Global function operator==

```
public IloPredicate< IloObject > operator==(IloEvaluator< IloObject > left,
IloEvaluator< IloObject > right)
public IloPredicate< IloObject > operator==(IloEvaluator< IloObject > left, IloNum
threshold)
public IloPredicate< IloObject > operator==(IloNum threshold, IloEvaluator<
IloObject > right)
```

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

Creates an equality predicate from two evaluators.
These operators create a new `IloPredicate<IloObject>` instance by comparing the value returned by an evaluator with either that of another evaluator or a given value. The semantics of the new predicate is an equality comparison. The first function creates a predicate which returns `IloTrue` if and only if the value returned by the left evaluator is equal to the value returned by the right evaluator. The second function creates a predicate which returns `IloTrue` if and only if the value returned by the left evaluator is equal to the value given as argument. The third function creates a predicate that returns `IloTrue` if and only if the value given as argument is equal to the value returned by the right evaluator.

For more information, see Selectors.

Global function operator==

```
public IloBool operator==(const IloIntervalList intervals1, const IloIntervalList
intervals2)
```

Definition file: ilconcert/ilointervals.h

Returns `IloTrue` for same interval lists.
This operator returns `IloTrue` if the interval lists are the same. That is, `IloTrue` is returned if they have the same definition interval and if they contain the same intervals. Note that it compares the content of the interval lists as well as the equality of implementation pointer. See also `IloIntervalList`.

Global function operator==

```
public IloBool operator==(const IloNumToAnySetStepFunction f1, const
IloNumToAnySetStepFunction f2)
```

Definition file: ilconcert/ilosetfunc.h

overloaded operator.
This operator returns `IloTrue` if the functions are the same. That is, `IloTrue` is returned if they have the same definition interval and if they have the same value over time. Note that it compares the content of the functions as well as the equality of implementation pointer. See also: `IloNumToAnySetStepFunction`.

Global function operator==

```
public IlcBool operator==(const IlcRevAny & rev, IlcAny value)
public IlcBool operator==(const IlcRevBool & rev, IlcBool bexp)
public IlcBool operator==(IlcBool bexp, const IlcRevBool & rev)
public IlcBool operator==(const IlcRevBool & rev1, const IlcRevBool & rev2)
public IlcBool operator==(const IlcRevInt & rev, IlcInt value)
public IlcBool operator==(IlcInt value, const IlcRevInt & rev)
public IlcBool operator==(const IlcRevInt & rev1, const IlcRevInt & rev2)
public IlcBool operator==(IlcAny value, const IlcRevAny & rev)
public IlcBool operator==(const IlcRevAny & rev1, const IlcRevAny & rev2)
public IlcBool operator==(const IlcRevFloat & rev, IlcFloat value)
public IlcBool operator==(IlcFloat value, const IlcRevFloat & rev)
public IlcBool operator==(const IlcRevFloat & rev1, const IlcRevFloat & rev2)
public IlcBool operator==(const IlcIntToIntStepFunction & f1, const
IlcIntToIntStepFunction & f2)
```

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>

This operator compares its arguments and returns `IlcTrue` if they are equal; otherwise, it returns `IlcFalse`.

See Also: IlcRevAny, IlcRevBool, IlcRevFloat, IlcRevInt, IlcIntToIntStepFunction

Global function operator==

```
public IlcConstraint operator==(const IlcIntExp exp1, const IlcIntExp exp2)
public IlcConstraint operator==(const IlcAnyExp exp1, const IlcAnyExp exp2)
public IlcConstraint operator==(const IlcAnyExp exp1, IlcAny exp2)
public IlcConstraint operator==(IlcAny exp1, const IlcAnyExp exp2)
public IlcConstraint operator==(const IlcIntExp exp1, IlcInt exp2)
public IlcConstraint operator==(IlcInt exp1, const IlcIntExp exp2)
public IlcConstraint operator==(IlcAnySetVar set1, IlcAnySetVar set2)
public IlcConstraint operator==(IlcAnySetVar set1, IlcAnySet set2)
public IlcConstraint operator==(IlcAnySet set1, IlcAnySetVar set2)
public IlcConstraint operator==(IlcIntSetVar set1, IlcIntSetVar set2)
public IlcConstraint operator==(IlcIntSet set1, IlcIntSetVar set2)
public IlcConstraint operator==(IlcIntSetVar set1, IlcIntSet set2)
public IlcConstraint operator==(const IlcFloatExp exp1, const IlcFloatExp exp2)
public IlcConstraint operator==(const IlcFloatExp exp1, IlcFloat exp2)
public IlcConstraint operator==(IlcFloat exp1, const IlcFloatExp exp2)
public IlcConstraint operator==(const IlcConstraint ct1, const IlcConstraint ct2)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This operator creates and returns an equality constraint between its arguments.

If one or both of its arguments are constrained set variables (instances of `IlcAnySetVar` or `IlcIntSetVar` or their subclasses), then when you post this constraint, the sets are replaced by their intersection. The constraint is stored so that any modification of one of those constrained set variables is propagated to the other.

If one or both of its arguments are constrained floating-point or integer variables, then when you post this constraint, it will be associated with the `whenRange` propagation event.

When both of its arguments are constraints (instances of `IlcConstraint`), then the constraint that this operator creates and returns forces its two arguments to be equivalent.

When you create a constraint, it has no effect until you post it.

See Also: IlcAnyExp, IlcAnySetVar, IlcConstraint, IlcFloatExp, IlcIntExp, IlcIntSetVar

Global function operator==

```
public IloConstraint operator==(IloNumExprArg base, IloNumExprArg expr)
public IloRange operator==(IloNumExprArg base, IloNum val)
public IloRange operator==(IloNum val, IloNumExprArg eb)
```

Definition file: ilconcert/ilolinear.h

Overloaded C++ operator.
This overloaded C++ operator constrains its two arguments to be equal. In order to be taken into account, this constraint must be added to a model and extracted for an algorithm.

Global function operator==

```
public IloBool operator==(const IloNumToNumStepFunction f1, const
IloNumToNumStepFunction f2)
```

Definition file: ilconcert/ilonumfunc.h

Overloaded operator tests equality of numeric functions.
This operator returns `IloTrue` if the functions `f1` and `f2` are the same. That is, `IloTrue` is returned if they have the same definition interval and if they have the same value over time. Note that it compares the content of the functions as well as the equality of implementation pointer. See also: `IloNumToNumStepFunction`.

Global function IloSplit

`public IloGoal` **`IloSplit`**`(const IloEnv env, const IloNumVarArray vars, IloBool increaseMinFirst=IloTrue, IloNum precision=0)`

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal in a Concert Technology model. The goal attempts to assign values to the constrained floating-point variables in the array `vars`. To do so, `IloSplit` scans the variables from the first to the last and repeats its procedure until all the variables are bound.

At each step of the scan, if the current variable is not yet bound, `IloSplit` sets a choice point, replaces the domain of the current variable by one of the halves, and examines the next variable.

For example, `IloSplit` starts with the first variable; if the first variable has not already been bound, `IloSplit` sets a choice point, then replaces the domain of that variable by one of the halves; `IloSplit` then examines the second variable. In contrast, if the first variable has already been bound, `IloSplit` examines the second variable directly.

When the argument `increaseMin` is `IloTrue`, the upper half of the domain is tried first. When `increaseMin` is `IloFalse`, the lower half of the domain is tried first.

When the optional argument `precision` is strictly positive, at each choice point, the domains of the variables are reduced more than usual. This reduction is the same as the one performed by the goal `IloGenerateBounds`.

This function returns an instance of `IloGoal` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. `IloSplit` is extracted to `IlcSplit`. An instance of `IloSolver` extracts the goal that it returns as an instance of `IlcGoal` for use during a Solver search.

This function works on numerical variables of type `Float` and type `Int`.

For an `IloNumVarArray`, IloSplit is extracted to `IlcSplit`. `IlcSplit` differs from `IlcGenerate` in the way the variables are chosen. `IlcSplit` considers first the first variable in the array then the second one and so on. When it reaches the end of the array it starts again from the beginning.

See Also: IloDichotomize, IloGoal, IlcSplit

Global function IlcSplit

```
public IlcGoal IlcSplit(const IlcFloatVarArray vars, IlcBool
increaseMinFirst=IlcTrue, IlcFloat precSolveBounds=0)
```

Definition file: ilsolver/nlinflt.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal, a primitive in the Solver algorithms that search for solutions. This goal binds each constrained variable in its argument `vars`.

`IlcSplit` scans the variables from the first to the last and repeats its procedure until all the variables are bound.

At each step of the scan, if the current variable is not yet bound, `IlcSplit` sets a choice point, replaces the domain of the current variable by one of the halves, and examines the next variable.

The optional argument `increaseMinFirst` must be a Boolean value, either `IlcTrue` or `IlcFalse`. If it is `IlcTrue`, then the upper half of the domain is tried first; otherwise, the lower half is tried first.

When the optional argument `precSolveBounds` is strictly positive, `IlcSolveBounds` is called before each choice point.

For example, `IlcSplit` starts with the first variable; if the first variable has not already been bound, `IlcSplit` sets a choice point, then replaces the domain of that variable by one of the halves; `IlcSplit` then examines the second variable. In contrast, if the first variable has already been bound, `IlcSplit` examines the second variable directly.

See Also: IlcBestGenerate, IlcDichotomize, IlcGenerate, IlcSolveBounds

Global function IloChangeValue

`public IloNHood` **`IloChangeValue`**`(IloEnv env, IloIntVarArray vars, IloInt min, IloInt max, const char * name=0)`

Definition file: ilsolver/iimnhood.h
Include file: <ilsolver/iimls.h>

This function returns a neighborhood that can be used to change the value of one variable in `vars` within the range `min` and `max`.

The neighborhood changes the value of a single variable in `vars` to a new value in the range `min` to `max`. Specifically, for each variable/value pair in `vars`, and in the range `[min, max]`, there is a neighbor in the neighborhood that sets the variable to that value. The optional argument `name`, if supplied, becomes the name of the returned neighborhood.

See Also: IloNHood, IloNHoodI

Global function IloDFSEvaluator

`public IloNodeEvaluator` **`IloDFSEvaluator`**`(const IloEnv env)`

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a *node evaluator* which implements depth-first search in a Concert Technology model.

This function returns an instance of `IloNodeEvaluator` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the node evaluator that it returns as an instance of `IlcNodeEvaluator` for use during a Solver search.

See Also: IloIDFSEvaluator, IlcNodeEvaluator, IloNodeEvaluator

Global function IlcAbs

```
public IlcIntExp IlcAbs(const IlcIntExp exp)
public IlcFloat IlcAbs(IlcFloat exp)
public IlcInt IlcAbs(IlcInt exp)
public IlcFloatExp IlcAbs(const IlcFloatExp exp)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function creates a new constrained expression equal to the absolute value of its constrained argument `exp`. The effects of this function are reversible.

If its argument is a numeric expression (for example, an ordinary floating-point number), this function simply returns the absolute value of its argument.

See Also: IlcFloatExp, IlcIntExp

Global function IlcAllDiffAggregator

`public IlcConstraintAggregator IlcAllDiffAggregator(IloSolver solver)`

Definition file: ilsolver/ilosolverhandle.h
Include file: <ilsolver/ilosolver.h>

This aggregator groups binary constraints of difference (`x != y`) and recognizes alldiff constraints as much as possible. It posts these constraints at extraction in place of every set of binary difference constraints that represent an alldiff constraint.

Global function IlcRandomVarEvaluator

```
public IloEvaluator< IlcIntVar > IlcRandomVarEvaluator(IloEnv env)
public IloEvaluator< IlcIntVar > IlcRandomVarEvaluator(IloSolver solver)
```

Definition file: ilsolver/custgoal.h
Include file: <ilsolver/ilosolver.h>

This function returns an instance of the evaluator `IloEvaluator<IlcIntVar>`. The evaluation returns a random value between 0 and 1.

Global function IlcSin

```
public IlcFloatExp IlcSin(const IlcFloatExp x)
public IlcFloat IlcSin(IlcFloat x)
```

Definition file: ilsolver/nlinflt.h
Include file: <ilsolver/ilosolver.h>

When its argument is a constrained floating-point expression, this function creates a constrained floating-point expression (that is, an instance of `IlcFloatExp` or one of its subclasses) which is equal to the sine of its argument `x` expressed in radians. The effects of this function are reversible.

When its argument is an unconstrained numeric value (that is, a value of type `IlcFloat`), this function returns the sine of its argument.

If you want to manipulate constrained floating-point expressions in degrees, we strongly recommend that you call the trigonometric functions on variables expressed in radians and then convert the results to degrees (rather than declaring the constrained floating-point expressions in degrees and then converting them to radians to call the trigonometric functions).

The reason for that advice is that the method we recommend gives more accurate results in the context of the usual floating-point pitfalls.

See Also: IlcArcCos, IlcArcSin, IlcArcTan, IlcCos, IlcDegToRad, IlcFloatExp, IlcHalfPi, IlcPi, IlcQuarterPi, IlcRadToDeg, IlcTan, IlcThreeHalfPi, IlcTwoPi

Global function IlcDegreeInformation

```
public IlcConstraintAggregator IlcDegreeInformation(IloSolver solver)
```

Definition file: ilsolver/ilosolverhandle.h
Include file: <ilsolver/ilosolver.h>

This aggregator maintains information about the dynamic degree of the domains of variables. It is required to use the functions:

```
IloInt IloSolver::getDegree(const IlcIntVar x) const;
IloInt IloSolver::getDegree(const IloIntVar x) const;
```

Global function IlcDegreeVarEvaluator

```
public IloEvaluator< IlcIntVar > IlcDegreeVarEvaluator(IloEnv env)
public IloEvaluator< IlcIntVar > IlcDegreeVarEvaluator(IloSolver solver)
```

Definition file: ilsolver/custgoal.h
Include file: <ilsolver/ilosolver.h>

This function returns an instance of the evaluator `IloEvaluator<IlcIntVar>`. The evaluation returns the result of the function call `solver.getDegree(x)`, where `x` is the evaluated variable.

Global function IlcEqAbstraction

```
public IlcConstraint IlcEqAbstraction(IlcAnyVarArray ys, IlcAnyVarArray xs,
IlcAnyArray vals, IlcAny abstractValue)
public IlcConstraint IlcEqAbstraction(IlcIntVarArray ys, IlcIntVarArray xs,
IlcIntArray vals, IlcInt abstractValue)
```

Definition file: ilsolver/ilcany.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a constraint that `ys[i]` is the abstraction of `xs[i]` over `vals` with respect to `abstractValue` for use during a Solver search.

The argument `xs` should be an array of constrained variables. The argument `vals` is an array of integers or pointers. The argument `abstractValue` is a value that must *not* belong to any of the variables in `xs` and must not appear in `vals`.

For each `xs[i]`, Solver creates a variable corresponding to the *abstraction* of `xs[i]` with respect to `vals`. In other words, for every variable in `xs`, `xs[i]`, Solver creates a variable `ys[i]` such that the domain of `ys[i]` is equal to the set made up of the value `abstractValue` plus all those values `xs[j]` that also belong to the array `vals`. Then internally Solver maintains these conditions:

- `xs[i]` is a value in the array `vals` if and only if `ys[i] == xs[i]`;
- `xs[i]` is *not* a value in the array `vals` if and only if `ys[i] == abstractValue`.

This function makes it easy to define constraints that impinge only on a particular set of values from the domains of constrained variables.

For a function that returns an array (rather than a constraint), see `IlcAbstraction`.

For a constraint suitable for use in a *model*, see `IloAbstraction`.

See Also: IlcAbstraction, IlcBoolAbstraction, IloAbstraction

Global function IloEqSum

```
public IloConstraint IloEqSum(const IloEnv, const IloIntSetVar var1, const
IloIntVar var2, const IloIntToIntFunction f)
public IloConstraint IloEqSum(const IloEnv, const IloIntSetVar var1, const
IloIntVar var2, const IloIntToIntVarFunction f)
```

Definition file: ilconcert/iloset.h

For IBM® ILOG® Solver: a constraint forcing a variable to the sum of returned values.
This function creates and returns a constraint (an instance of IloConstraint) for use in a model. The constraint forces `var2` to the sum of the values returned by the function `f` when it is applied to the variable `var1`.

In order for the constraint to take effect, you must add it to a model with the template IloAdd or the member function IloModel::add and extract the model for an algorithm with the member function IloAlgorithm::extract.

Global function IloRandomize

```
public IloNHood IloRandomize(IloEnv env, IloNHood nhood, IloRandom rand, const char
* name=0)
```

Definition file: ilsolver/iimnhood.h
Include file: <ilsolver/iimls.h>

This function returns a neighborhood that behaves as `nhood`, except that the neighborhood order is randomly jumbled each time that the returned neighborhood is started. The argument `rand` is used to generate the required random numbers. The optional argument `name`, if provided, becomes the name of the returned neighborhood. The returned neighborhood uses memory linear in the size of the neighborhood `nhood` to store the randomization order.

This type of randomized neighborhood is extremely useful in local search procedures, as neighbors with low indices are often preferred to those with high indices simply because they are examined first. This can lead to a stagnation in the search process, which can be avoided with the use of a randomized neighborhood.

See Also: IloRandom, IloNHood, IloNHoodI

Global function IlcSizeVarEvaluator

```
public IloEvaluator< IlcIntVar > IlcSizeVarEvaluator(IloEnv env)
public IloEvaluator< IlcIntVar > IlcSizeVarEvaluator(IloSolver solver)
```

Definition file: ilsolver/custgoal.h
Include file: <ilsolver/ilosolver.h>

This function returns an instance of the evaluator `IloEvaluator<IlcIntVar>`. The evaluation returns the result of the function call `x.getSize()`, where `x` is the evaluated variable.

Global function operator!=

```
public IlcBool operator!=(const IlcRevAny & rev, IlcAny value)
public IlcBool operator!=(const IlcRevBool & rev, IlcBool bexp)
public IlcBool operator!=(IlcBool bexp, const IlcRevBool & rev)
public IlcBool operator!=(const IlcRevBool & rev1, const IlcRevBool & rev2)
public IlcBool operator!=(const IlcRevInt & rev, IlcInt value)
public IlcBool operator!=(IlcInt value, const IlcRevInt & rev)
public IlcBool operator!=(const IlcRevInt & rev1, const IlcRevInt & rev2)
public IlcBool operator!=(IlcAny value, const IlcRevAny & rev)
public IlcBool operator!=(const IlcRevAny & rev1, const IlcRevAny & rev2)
public IlcBool operator!=(const IlcRevFloat & rev, IlcFloat value)
public IlcBool operator!=(IlcFloat value, const IlcRevFloat & rev)
public IlcBool operator!=(const IlcRevFloat & rev1, const IlcRevFloat & rev2)
```

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>

This operator compares its arguments and returns `IlcTrue` if they are *not* equal; otherwise, it returns `IlcFalse`.

See Also: IlcRevAny, IlcRevBool, IlcRevFloat, IlcRevInt

Global function operator!=

```
public IlcConstraint operator!=(const IlcIntExp exp1, const IlcIntExp exp2)
public IlcConstraint operator!=(const IlcAnyExp exp1, const IlcAnyExp exp2)
public IlcConstraint operator!=(const IlcAnyExp exp1, IlcAny exp2)
public IlcConstraint operator!=(IlcAny exp1, const IlcAnyExp exp2)
public IlcConstraint operator!=(const IlcIntExp exp1, IlcInt exp2)
public IlcConstraint operator!=(IlcInt exp1, const IlcIntExp exp2)
public IlcConstraint operator!=(IlcAnySetVar set1, IlcAnySetVar set2)
public IlcConstraint operator!=(IlcAnySetVar set1, IlcAnySet set2)
public IlcConstraint operator!=(IlcAnySet set1, IlcAnySetVar set2)
public IlcConstraint operator!=(IlcIntSetVar set1, IlcIntSetVar set2)
public IlcConstraint operator!=(IlcIntSet set1, IlcIntSetVar set2)
public IlcConstraint operator!=(IlcIntSetVar set1, IlcIntSet set2)
public IlcConstraint operator!=(const IlcConstraint ct1, const IlcConstraint ct2)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This operator creates and returns an inequality constraint between its arguments.

When one or both of its arguments are constrained integer expressions, then when you post it, this constraint is associated with the `whenRange` propagation event.

When both of its arguments are constraints (instances of `IlcConstraint`), the constraint that this operator creates and returns is the exclusive disjunction of its two arguments. That is, the two arguments will be different from each other.

When you create a constraint, it has no effect until you post it.

See Also: IlcAnyExp, IlcAnySetVar, IlcConstraint, IlcIntExp, IlcIntSetVar, operator==, operator<=, operator>=

Global function operator!=

```
public IloDiff operator!=(IloNumExprArg arg1, IloNumExprArg arg2)
public IloConstraint operator!=(const IloAnyVar var1, const IloAnyVar var2)
public IloConstraint operator!=(const IloAnyVar var1, IloAny val)
public IloConstraint operator!=(IloAny val, const IloAnyVar var1)
public IloConstraint operator!=(const IloAnySetVar var1, const IloAnySetVar var2)
public IloConstraint operator!=(const IloAnySetVar var1, const IloAnySet set)
public IloConstraint operator!=(const IloAnySet set, const IloAnySetVar var1)
public IloDiff operator!=(IloNumExprArg arg, IloNum val)
public IloDiff operator!=(IloNum val, IloNumExprArg arg)
public IloConstraint operator!=(const IloIntSetVar var1, const IloIntSetVar var2)
public IloConstraint operator!=(const IloIntSetVar var, const IloIntSet set)
public IloConstraint operator!=(const IloIntSet set, const IloIntSetVar var)
```

Definition file: ilconcert/ilomodel.h

Overloaded C++ operator.
This overloaded C++ operator constrains its two arguments to be unequal (that is, different from each other). In order to be taken into account, this constraint must be added to a model and extracted for an algorithm.

Global function operator!=

```
public IloPredicate< IloObject > operator!=(IloEvaluator< IloObject > left,
IloEvaluator< IloObject > right)
public IloPredicate< IloObject > operator!=(IloEvaluator< IloObject > left, IloNum
threshold)
public IloPredicate< IloObject > operator!=(IloNum threshold, IloEvaluator<
IloObject > right)
```

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

Creates a non-equality predicate from two evaluators.
These operators create a new IloPredicate<IloObject> instance by comparing the value returned by an evaluator with either that of another evaluator or a threshold value. The semantics of the new predicate is a non-equality comparison. The first function creates a predicate which returns IloTrue if and only if the value returned by the left evaluator is not equal to the value returned by the right evaluator. The second function creates a predicate which returns IloTrue if and only if the value returned by the left evaluator is not equal to the threshold value. The third function creates a predicate that returns IloTrue if and only if the threshold value is not equal to the value returned by the right evaluator.

For more information, see Selectors.

Global function IloDestroyAll

`public IloPoolProc` **`IloDestroyAll`**`(IloEnv env)`

Definition file: ilsolver/iimiloproc.h
Include file: <ilsolver/iim.h>

Creates a processor which destroys supplied pool elements.
This function creates a processor which destroys supplied pool elements in the allocation environment `env`.

The following code shows a replacement goal where the entire population is destroyed and replaced by the contents of the offspring pool:

```
// Replace the entire population with offspring
IloGoal replacementGoal =
  IloExecuteProcessor(env, population >> IloDestroyAll(env)) &&
  IloExecuteProcessor(env, offspring >> population);
```

Global function operator/

```
public IloNumExprArg operator/(const IloNumExprArg x, const IloNumExprArg y)
public IloNumExprArg operator/(const IloNumExprArg x, IloNum y)
public IloNumExprArg operator/(IloNum x, const IloNumExprArg y)
```

Definition file: ilconcert/iloexpression.h

Returns an expression equal to the quotient of its arguments.
This overloaded C++ operator returns an expression equal to the quotient of its arguments. Its arguments may be numeric values or numeric variables. For integer division, use IloDiv.

Global function operator/

```
public IlcFloatExp operator/(const IlcFloatExp exp1, IlcFloat exp2)
public IlcIntExp operator/(const IlcIntExp exp1, IlcInt exp2)
public IlcIntExp operator/(IlcInt exp1, const IlcIntExp exp2)
public IlcIntExp operator/(const IlcIntExp exp1, const IlcIntExp exp2)
public IlcFloatExp operator/(IlcFloat exp1, const IlcFloatExp exp2)
public IlcFloatExp operator/(const IlcFloatExp exp1, const IlcFloatExp exp2)
```

Definition file: ilsolver/linfloat.h
Include file: <ilsolver/ilosolver.h>

This arithmetic operator divides its first argument by its second. It has been overloaded to handle constrained expressions appropriately. The domain of the resulting expression is computed from the domains of the combined expressions as you would expect. If the domains of the dividend and divisor include 0 (zero), then 0/0 is supported; it does not constrain the resulting expression.

See Also: IlcFloatExp, IlcIntExp

Global function operator/

```
public IloEvaluator< IloObject > operator/(IloEvaluator< IloObject > left,
IloEvaluator< IloObject > right)
public IloEvaluator< IloObject > operator/(IloEvaluator< IloObject > left, IloNum
c)
public IloEvaluator< IloObject > operator/(IloNum c, IloEvaluator< IloObject >
right)
```

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

These operators create a composite `IloEvaluator<IloObject>` instance. The semantics of the new evaluator are the division of the values of the component evaluators. The first function combines two evaluators, dividing the value returned by the first by the value returned by the second. The other two functions combine an `IloNum` value with an evaluator. In the first of these functions, the evaluator is the dividend and the `IloNum` value is the divisor, while in the second of these functions, those roles are reversed.

For more information, see Selectors.

Global function IloAddConstraint

```
public IloGoal IloAddConstraint(const IloConstraint constraint)
```

Definition file: ilsolver/ilosolveri.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal for use in a search. This goal will extract the `constraint` for a solver (an instance of `IloSolver`) during a search in a *reversible* way. In other words, it will be possible for a solver to backtrack and cancel the constraint during the search. In particular, if the solver backtracks before the execution of this goal, then it will also backtrack the extraction of the constraint and its addition to the model.

See Also: IloSolver

Global function IloDichotomize

```
public IloGoal IloDichotomize(const IloEnv env, const IloNumVarArray vars, const
IloChooseFloatIndex=IloChooseFirstUnboundFloat, IloBool increaseMin=IloTrue, IloNum
precision=0)
public IloGoal IloDichotomize(const IloEnv env, const IloNumVar var, IloBool
increaseMin=IloTrue, IloNum precision=0)
```

Definition file: ilsolver/ilosolverfloat.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal in a Concert Technology model. The goal attempts to assign values to the constrained floating-point variables in the array `vars`. To do so, it recursively searches half the domain of each variable in the array at a time.

If the current variable is not yet bound, `IloDichotomize` sets a choice point, replaces the complete domain of the variable by one half of the domain and calls itself recursively. If failure occurs then, it replaces the domain of the current variable by the other half, and tries again recursively.

`IloDichotomize` chooses the order in which to try to bind variables in the array based on the argument of type `IlcChooseFloatIndex`. If no choice criterion is passed, `IloDichotomize` will recursively consider the first variable in the array until it is instantiated and then proceed to the next one.

When the argument `increaseMin` is `IloTrue`, the upper half of the domain is tried first. When `increaseMin` is `IloFalse`, the lower half of the domain is tried first.

When the optional argument `precision` is strictly positive, at each choice point, the domains of the variables are reduced more than usual. This reduction is the same as the one performed by the goal `IloGenerateBounds`.

This function returns an instance of `IloGoal` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. `IloDichotomize` is extracted to `IlcGenerate`. An instance of `IloSolver` extracts the goal that it returns as an instance of `IlcGoal` for use during a Solver search.

This function works on numerical variables of type `Float` and type `Int`.

For an `IloNumVarArray` of either type `Float` or type `Int`, `IloDichotomize` is extracted to `IlcGenerate`. `IlcGenerate` takes a variable (chosen by `_choose`) and if uninstantiated considers the two halves of the domain (according to `increaseMinFirst`. It then repeats this. If no `_choose` selector is given it will recursively consider the first variable until it is instantiated and then proceed to the next one.

For an `IloNumVar` of either type `Float` or type `Int`, `IloDichotomize` is extracted to `IlcInstantiate`. `IlcInstantiate` takes a variable (chosen by `_choose`) and if uninstantiated considers the two halves of the domain (according to `increaseMinFirst`. It then repeats this. If no `_choose` selector is given it will recursively consider the first variable until it is instantiated and then proceed to the next one.

See Also: IloGoal, IloSplit, IlcDichotomize, IlcGenerate, IlcInstantiate

Global function IlcMinimizeVar

```
public IlcSearchSelector IlcMinimizeVar(IloSolver solver, IlcIntVar v, IlcInt
step=1)
public IlcSearchSelector IlcMinimizeVar(IloSolver solver, IlcFloatVar v, IlcFloat
step=1e-4)
```

Definition file: ilsolver/search.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a search selector that implements a minimization search for the variable `v`.

The search selector that this function creates and returns does several things:

- It stores the leaf of the search tree corresponding to the optimal value of the variable `v` and then reactivates this variable after the complete exploration of the search tree defined by the goal passed to `IloSelectSearch`.
- It manages the upper bound on the objective variable. As soon as a solution of value `d` is found, the constraint `v <= d - step` is added to the solver for the remainder of the search tree.
- Open nodes are evaluated. The evaluation of an open node is equal to the current minimum of the variable v when the node is created. When the solver asks for an open node, it checks whether the current upper bound on the objective is less than the evaluation of the node. If so, the node is safely discarded.

See Also: IlcSearchSelector, IloFirstSolution, IloSelectSearch

Global function IloGoalFail

`public IloGoal IloGoalFail(const IloEnv env, IloAny label=0)`

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal that always fails. It may be useful when you are searching for all possible ways to execute another goal. The idea is to try all possible subgoals of all choice points by calling this function after each successful execution of the other goal, thus causing backtracking in an instance of `IloSolver` and launching the search for another execution. You may optionally provide the `label` of a choice point as an argument to this function.

When it takes an instance of the class `IloEnv` as a parameter, it returns an instance of `IloGoal` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the goal that it returns as an instance of `IlcGoal` for use during a Solver search.

See Also: IloGoal, IlcGoalFail

Global function operator<=

```
public IloRange operator<=(const IloRangeBase base, IloNum val)
```

Definition file: ilconcert/ilolinear.h

This overloaded C++ operator constrains its first argument to be less than or equal to its second argument. In order to be taken into account, this constraint must be added to a model and extracted for an algorithm.

Global function operator<=

```
public IloPredicate< IloObject > operator<=(IloEvaluator< IloObject > left,
IloEvaluator< IloObject > right)
public IloPredicate< IloObject > operator<=(IloEvaluator< IloObject > left, IloNum
threshold)
public IloPredicate< IloObject > operator<=(IloNum threshold, IloEvaluator<
IloObject > right)
```

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

Creates a less-than-or-equal predicate from two evaluators.
These operators create a new `IloPredicate<IloObject>` instance by comparing the value returned by an evaluator with either that of another evaluator or a threshold value. The semantics of the new predicate is a less-than-or-equal comparison. The first function creates a predicate which returns `IloTrue` if and only if the value returned by the left evaluator is less than or equal to the value returned by the right evaluator. The second function creates a predicate which returns `IloTrue` if and only if the value returned by the left evaluator is less than or equal to the threshold value. Finally, the third function creates a predicate that returns `IloTrue` if and only if the threshold value is less than or equal to the value returned by the right evaluator.

For more information, see Selectors.

Global function operator<=

```
public IloConstraint operator<=(IloNumExprArg base, IloNumExprArg base2)
public IloRange operator<=(IloNumExprArg base, IloNum val)
public IloRangeBase operator<=(IloNum val, const IloNumExprArg expr)
```

Definition file: ilconcert/ilolinear.h

overloaded C++ operator
This overloaded C++ operator constrains its first argument to be less than or equal to its second argument. In order to be taken into account, this constraint must be added to a model and extracted for an algorithm.

Global function operator<=

```
public IlcBool operator<=(const IlcRevFloat & rev, IlcFloat value)
public IlcBool operator<=(const IlcRevInt & rev, IlcInt value)
public IlcBool operator<=(IlcInt value, const IlcRevInt & rev)
public IlcBool operator<=(const IlcRevInt & rev1, const IlcRevInt & rev2)
public IlcBool operator<=(IlcFloat value, const IlcRevFloat & rev)
public IlcBool operator<=(const IlcRevFloat & rev1, const IlcRevFloat & rev2)
public IlcBool operator<=(const IlcIntToIntStepFunction & f1, const
IlcIntToIntStepFunction & f2)
```

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>

This operator compares its arguments; if the first argument is less than or equal to the second, then it returns `IlcTrue`; otherwise, it returns `IlcFalse`.

See Also: IlcRevFloat, IlcRevInt, IlcIntToIntStepFunction

Global function operator<=

```
public IlcConstraint operator<=(const IlcIntExp exp1, const IlcIntExp exp2)
public IlcConstraint operator<=(const IlcIntExp exp1, IlcInt exp2)
public IlcConstraint operator<=(IlcInt exp1, const IlcIntExp exp2)
public IlcConstraint operator<=(const IlcFloatExp exp1, const IlcFloatExp exp2)
public IlcConstraint operator<=(const IlcFloatExp exp1, IlcFloat exp2)
public IlcConstraint operator<=(IlcFloat exp1, const IlcFloatExp exp2)
public IlcConstraint operator<=(const IlcConstraint ct1, const IlcConstraint ct2)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This operator creates and returns an inequality constraint between its arguments; that is, the first argument must be less than or equal to the second.

When its arguments are constraints (instances of `IlcConstraint`), the constraint that this operator creates and returns is the implication between its arguments: `ct1` implies `ct2`. The implication operator may look strange. This operator was selected for two reasons. First, there is no operator in C++ that looks like the conventional implication sign, =>. One possible candidate is -> but because of its purpose in C++, that choice is too dangerous. Second, if you recall that 1 represents `IlcTrue`, and 0 represents `IlcFalse`, then implication is the "less than or equal to" constraint.

This constraint is associated with the `whenRange` propagation event after you post it if its arguments are constrained floating-point expressions or constrained integer expressions.

When you create a constraint, it has no effect until you post it.

See Also: IlcConstraint, IlcFloatExp, IlcIntExp, IlcLeOffset, operator>=, operator!=, operator==

Global function IlcImpactValueEvaluator

```
public IloEvaluator< IlcInt > IlcImpactValueEvaluator(IloEnv env)
public IloEvaluator< IlcInt > IlcImpactValueEvaluator(IloSolver solver)
```

Definition file: ilsolver/custgoal.h
Include file: <ilsolver/ilosolver.h>

This function returns an instance of the evaluator `IloEvaluator<IlcInt>`. The evaluation returns the result of the function `solver.getImpact(x, a)`, where `x` is the selected variable given in context and `a` is the value evaluated.

Global function IlcChooseMinSizeAnySet

```
public IlcInt IlcChooseMinSizeAnySet(const IlcAnySetVarArray vars)
```

Definition file: ilsolver/ilcset.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the smallest domain from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseAnySetIndex, IloGenerate

Global function IlcChooseFirstUnboundFloat

`public IlcInt **IlcChooseFirstUnboundFloat**(const IlcFloatVarArray vars)`

Definition file: ilsolver/linfloat.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the first unbound constrained variable that it encounters in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseFloatIndex, IlcChooseFloatIndex1, IlcChooseFloatIndex2, IlcFloatVarArray, IloGenerate, IlcBoolVarArray

Global function IlcChooseMaxMinInt

```
public IlcInt IlcChooseMaxMinInt(const IlcIntVarArray vars)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the greatest minimum from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseIntIndex, IloGenerate

Global function IloChooseMaxSizeInt

```
public IlcInt IloChooseMaxSizeInt(const IlcIntVarArray vars)
```

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the greatest domain from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseIntIndex, IloGenerate

Global function IloExecuteProcessor

```
public IloGoal IloExecuteProcessor(IloEnv env, IloPoolProc proc, const char *
name=0)
```

Definition file: ilsolver/iimoperator.h
Include file: <ilsolver/iim.h>

Casts a pool processor into an `IloGoal` for execution via an instance of `IloSolver`.
This function returns a goal that casts a pool processor into an `IloGoal` for execution using an instance of `IloSolver`. The parameter `proc` is the processor to be converted and the parameter `name` defines the name of the newly created goal. The goal is allocated on environment `env`

Global function IlcPower

```
public IlcFloatExp IlcPower(const IlcFloatExp x, const IlcFloat p)
public IlcFloatExp IlcPower(const IlcFloatExp x, const IlcInt p)
public IlcFloatExp IlcPower(IlcFloatExp x, IlcFloatExp p)
public IlcFloatExp IlcPower(IlcFloat x, IlcFloatExp p)
public IlcFloat IlcPower(IlcFloat x, IlcFloat p)
```

Definition file: ilsolver/nlinflt.h
Include file: <ilsolver/ilosolver.h>

This function creates a new constrained expression equal to the base *x* raised to the power *p*, that is, *xp*. If *p* is a floating-point value, then *x* is constrained to be greater than or equal to 0 (zero). The argument `p` must be different from 0 (zero). The effects of this function are reversible.

If the arguments are simply floating-point numbers (that is, they are not constrained expressions), it merely returns an instance of `IlcFloat` equal to the base raised to the power. The base should be greater than 0 (zero).

The distinction between the two versions (whether the exponent is integer or floating-point) of this function is not always handled correctly by certain C++ compilers. To cope with their vagaries, if you want to use the integer exponent version, add `L` (for `long`) after the exponent, like this:

```
IlcPower(x, 19L);
```

See Also: IlcExponent, IlcFloatExp, IlcMonotonicDecreasingFloatExp, IlcMonotonicIncreasingFloatExp, IlcSquare

Global function IlcAllDiff

```
public IlcConstraint IlcAllDiff(const IlcAnyVarArray array)
public IlcConstraint IlcAllDiff(const IlcAnyVarArray array, IlcFilterLevel level)
public IlcConstraint IlcAllDiff(const IlcIntVarArray array, IlcFilterLevel level)
public IlcConstraint IlcAllDiff(const IlcIntVarArray array)
```

Definition file: ilsolver/ilcany.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a constraint stating that the constrained enumerated variables of `array` must take different values from each other when they are bound. In other words, this constraint extends the operator `!=` for an array of constrained variables. This constraint is for use during a Solver search, for example, inside a goal (an instance of `IlcGoal`) or inside a constraint (an instance of `IlcConstraint`). If you are looking for similar functionality for use in an IBM® ILOG® Concert Technology *model*, see `IloAllDiff`.

If you do not explicitly state a filter level, then Solver uses the default filter level for this constraint (that is, `IlcBasic`). The optional argument `level` may take one of the values of the enumeration `IlcFilterLevel`, with the exception of the value `IlcLow`. See `IlcFilterLevel` for an explanation of filter levels and their effect on constraint propagation.

You must post the returned constraint in order for it to be taken into account, for example, by adding it to an instance of `IloSolver`.

Example

For example, if you are looking for a Hamiltonian path in a graph that contains no directed cycles, (that is, you are looking for a path that visits each node exactly once) then `IlcAllDiff` with the parameter `IlcExtended` applied to an array consisting of the next nodes in the graph will produce the best propagation. In fact, it will achieve arc consistency in the search.

For more information, see `IloAllDiff`.

See Also: IlcAllNullIntersect, IlcAnyVarArray, IlcFilterLevel, IlcIntVarArray, IloAllDiff, operator!=

Global function IloSetMax

```
public IloGoal IloSetMax(const IloEnv env, const IloNumVar var, IloNum value)
```

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal for the constrained numeric variable `var` with `value` as its maximum.

When it takes an instance of the class `IloEnv` as a parameter, it returns an instance of `IloGoal` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the goal that it returns as an instance of `IlcGoal` for use during a Solver search.

For more information, see `IloNullIntersect`.

This function works on numerical variables of type `Float` and type `Int`.

See Also: IloGoal, IlcSetMax, IloNullIntersect

Global function IloChooseMaxRegretMin

```
public IlcInt IloChooseMaxRegretMin(const IlcIntVarArray vars)
```

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the maximal difference between the minimal possible value and the next minimal possible value from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is based on the principle of *least regret*. Regret is the difference between what would have been the best possible decision in a scenario and what was the actual decision.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseIntIndex, IloGenerate

Global function IloAllNullIntersect

```
public IloConstraint IloAllNullIntersect(const IloEnv env, const IloAnySetVarArray
vars)
public IloConstraint IloAllNullIntersect(const IloEnv env, const IloIntSetVarArray
vars)
```

Definition file: ilconcert/iloanyset.h

For IBM® ILOG® Solver: a constraint forcing one set to have no elements in common with another set. This function creates and returns a constraint (an instance of `IloConstraint`) for use in a model. The constraint makes sure that for the sets in the array `vars`, the intersection of `vars[i]` with `vars[j]` will be empty for all `i` and `j` when this constraint is satisfied.

In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

What Is Extracted

All the variables of the set or array that have been added to the model and that have not been removed from it will be extracted when the algorithm `IloSolver` (documented in the *IBM ILOG Solver Reference Manual*) extracts the constraint.

`IloCplex` does **not** extract this constraint.

Global function IloChooseMinSizeInt

`public IlcInt **IloChooseMinSizeInt**(const IlcIntVarArray vars)`

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the smallest domain from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseIntIndex, IloGenerate

Global function IlcSuccessRateVarEvaluator

```
public IloEvaluator< IlcIntVar > IlcSuccessRateVarEvaluator(IloEnv env)
public IloEvaluator< IlcIntVar > IlcSuccessRateVarEvaluator(IloSolver solver)
```

Definition file: ilsolver/custgoal.h
Include file: <ilsolver/ilosolver.h>

This function returns an instance of the evaluator `IloEvaluator<IlcIntVar>`. The evaluation returns the result of the function `solver.getSuccessRate(x)`, where `x` is the evaluated variable.

Global function IloChooseMaxMaxInt

```
public IlcInt IloChooseMaxMaxInt(const IlcIntVarArray vars)
```

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the greatest maximum from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseIntIndex, IloGenerate

Global function IloEqIntersection

```
public IloConstraint IloEqIntersection(const IloEnv env, const IloAnySetVar var1,
const IloAnySetVar var2, const IloAnySetVar var3)
public IloConstraint IloEqIntersection(const IloEnv env, const IloIntSetVar var1,
const IloIntSetVar var2, const IloIntSetVar var3)
```

Definition file: ilconcert/iloanyset.h

For IBM® ILOG® Solver: a constraint forcing the intersection of two sets to the elements of a third set.
This function creates and returns a constraint (an instance of `IloConstraint`) for use in Concert Technology.
The constraint forces the intersection of the sets `var2` and `var3` to be precisely the elements of the set `intersection`.

In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

Global function IloBoolAbstraction

`public IloConstraint` **`IloBoolAbstraction`**`(const IloEnv env, const IloBoolVarArray y, const IloIntVarArray x, const IloIntArray values)`
`public IloConstraint` **`IloBoolAbstraction`**`(const IloEnv env, const IloBoolVarArray avars, const IloAnyVarArray vars, const IloAnyArray values)`

Definition file: ilconcert/ilomodel.h

For constraint programming: creates a constraint to abstract an array of Boolean variables.
This function creates and returns a constraint that abstracts an array of constrained Boolean variables in a model. It differs from `IloAbstraction` in that its y-array is an array of Boolean variables (also known as 0-1 variables or binary variables). Like `IloAbstraction`, for each element `x[i]`, there is a variable `y[i]` corresponding to the abstraction of x[i] with respect to an array of values. That is,

```
x[i] = v with v in values if and only if y[i] = IloTrue;

x[i] = v with v not in values if and only if y[i] = IloFalse.
```

This constraint maintains a many-to-one mapping that makes it possible to define constraints that impinge only on a particular set of values from the domains of constrained variables.

Example

For simplicity, assume that an array `x` consists of three elements with the domains {3}, {4}, and {5}. Assume that the values we are interested in are {4, 8, 12, 16}. Then `IloBoolAbstraction` produces the elements of the array `y`, like this:

```
 X      &         Values              Y
---               -------          -------
 3                   4             IloFalse
 4                   8    -->      IloTrue
 5                  12             IloFalse
                    16
```

Adding a Constraint to a Model, Extracting a Model for an Algorithm

In order for the constraint returned by `IloBoolAbstraction` to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

Exceptions

If the arrays `x` and `y` are not the same size, this function throws the exception `IloBoolAbstraction::InvalidArraysException`.

Global function IlcAnd

```
public IlcGoal IlcAnd(const IlcGoal g1, const IlcGoal g2)
public IlcGoal IlcAnd(const IlcGoal g1, const IlcGoal g2, const IlcGoal g3)
public IlcGoal IlcAnd(const IlcGoal g1, const IlcGoal g2, const IlcGoal g3, const
IlcGoal g4)
public IlcGoal IlcAnd(const IlcGoal g1, const IlcGoal g2, const IlcGoal g3, const
IlcGoal g4, const IlcGoal g5)
```

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>

A goal can be defined in terms of other goals, called its *subgoals*. The function `IlcAnd` creates a goal composed of a sequence of other goals (between two and five subgoals). Executing this goal executes the subgoals from left to right. That apparent limitation of five subgoals can be overcome by several calls to the function, since `IlcAnd` is associative.

If a goal is null (that is, if its implementation is null), it will be silently ignored.

Examples:

First we'll define a goal, `PrintX`, like this:

```
ILCGOAL1(PrintX, IlcInt, value){
  IloSolver s = getSolver();
  s.out() << "PrintX: a goal with one data member" << endl;
  s.out() << value << endl;
  return 0;
}
```

Then the following statements:

```
s.solve(IlcAnd(PrintX(1), PrintX(2), PrintX(3));
```

produce the following output:

```
PrintX: executing a goal with one data member
1
PrintX: executing a goal with one data member
2
PrintX: executing a goal with one data member
3
```

Here's how to define a choice point with eight subgoals:

```
IlcAnd(IlcAnd(g1, g2, g3, g4, g5),
       IlcAnd(g6, g7, g8));
```

For more information, see the concept Goal.

See Also: IlcGoal, IlcOr

Global function IlcEqPartition

```
public IlcConstraint IlcEqPartition(IlcIntSetVar set, IlcIntSetVarArray vars)
public IlcConstraint IlcEqPartition(IlcAnySetVar set, IlcAnySetVarArray vars)
```

Definition file: ilsolver/ilcset.h
Include file: <ilsolver/ilosolver.h>

These functions create and return a constraint that forces the variables `vars` to be a partition of the variable `set`. `vars` are a partition of `set` if their union is equal to `set` and if their intersection is null. The variable set and all the variables in `vars` must be built from the same initial array.

See Also: IlcAnySetVar, IlcAnySetVarArray, IlcEqUnion, IlcIntSetVar, IlcIntSetVarArray

Global function IlcIfThen

`public void **IlcIfThen**(const IlcConstraint ct1, const IlcConstraint ct2)`

Definition file: ilsolver/numi.h
Include file: <ilsolver/ilosolver.h>

This function constrains its first argument to imply its second argument. That is, if `ct1` is satisfied, then `ct2` will be posted; if `ct2` is violated, then the opposite of `ct1` will be posted. The effects of this function are reversible.

This function is appropriate for use only during a Solver search (that is, inside a constraint or goal). If you are looking for similar functionality as a constraint to add to a model, consider the function `IloIfThen`.

Implementation

Inside a goal or constraint, this function is equivalent to these lines:

```
void IlcIfThen(IlcConstraint ct1, IlcConstraint ct2){
    IloSolver s = ct1.getSolver();
    s.add(!ct1 || ct2);
}
```

For more information, see `IloIfThen` in the *Concert Technology Reference Manual*.

See Also: IlcConstraint, IloIfThen

Global function IloNullIntersect

```
public IloConstraint IloNullIntersect(const IloEnv env, const IloAnySetVar var1,
const IloAnySetVar var2)
public IloConstraint IloNullIntersect(const IloEnv env, const IloAnySet var1, const
IloAnySetVar var2)
public IloConstraint IloNullIntersect(const IloEnv env, const IloAnySetVar var1,
const IloAnySet var2)
public IloConstraint IloNullIntersect(const IloEnv, const IloIntSetVar var1, const
IloIntSetVar var2)
public IloConstraint IloNullIntersect(const IloEnv, const IloIntSet var1, const
IloIntSetVar var2)
public IloConstraint IloNullIntersect(const IloEnv, const IloIntSetVar var1, const
IloIntSet var2)
```

Definition file: ilconcert/iloanyset.h

For IBM® ILOG® Solver: a constraint forcing one set to have no elements in common with another set.
This function creates and returns a constraint (an instance of `IloConstraint`) for use in a model. The constraint forces the set `var1` to have no elements in common with the set `var2`. In other words, the intersection of `var1` with `var2` will be empty when this constraint is satisfied.

In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

Global function IloEndMT

```
public void IloEndMT()
```

Definition file: ilconcert/iloenv.h

Ends multithreading.
This function ends multithreading in a Concert Technology application.

Global function operator>>

```
public istream & operator>>(istream & in, IloNumArray & a)
public istream & operator>>(istream & in, IloIntArray & a)
```

Definition file: ilconcert/iloenv.h

Overloaded C++ operator redirects input.
This overloaded C++ operator directs input to an input stream.

Global function operator>>

```
public IloPoolProc operator>>(IloPoolProc producer, IloPoolProc consumer)
```

Definition file: ilsolver/iimiloproc.h
Include file: <ilsolver/iim.h>

Returns a chain of two pool processors.
This chaining operator `>>` connects two pool processors such that the output of the first processor is connected to the input of the second.

When the composite processor is asked to produce output, it passes this request directly to the `consumer` processor. The consumer will then pass a request for input to the `producer`. The producer will then ask for input from any processor which may afterwards be connected to its left. On receiving the input the producer will process it, and pass the output to the consumer. The consumer will then in turn process its input and pass the result into its output pool where it can be picked up by the original processor who asked for the composite operator to produce output.

Global function IlcLimitSearch

`public IlcGoal IlcLimitSearch(IlcGoal goal, IlcSearchLimit searchLimit)`

Definition file: ilsolver/search.h
Include file: <ilsolver/ilosolver.h>

This function returns a goal that limits the exploration of the search tree defined by `goal` with the limit indicated by `searchLimit`. All nodes explored after that limit has been met are discarded.

See Also: IlcSearchLimit

Global function IlcMTMinimizeVar

```
public IlcSearchSelector IlcMTMinimizeVar(IloSolver solver, IlcIntVar v, IlcInt
step=1)
public IlcSearchSelector IlcMTMinimizeVar(IloSolver solver, IlcFloatVar v, IlcFloat
step=1e-4)
```

Definition file: ilsolver/search.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a search selector that implements a minimization search for *multithreaded* search according to the variable `v`.

If a variable `v` appears in more than one agent (that is, more than one worker), then every copy of the variable `v` must have the same name.

The search selector that this function creates and returns does several things:

- It stores the leaf of the search tree corresponding to the optimal value of the variable `v` and then reactivates this variable after the complete exploration of the search tree defined by the goal passed to `IlcSelectSearch`.
- It manages the upper bound on the objective variable. As soon as a solution of value `d` is found, the constraint `v <= d - step` is added to the solver for the remainder of the search tree.
- Open nodes are evaluated. The *evaluation* of an open node is equal to the current minimum of the variable v when the node is created. When the solver asks for an open node, it checks whether the current upper bound on the objective is less than the evaluation of the node. If so, the node is safely discarded.

Implementation

Here we describe important parts of the implementation of this function because it returns an instance of `IlcSearchSelector`. If you implement your own search selector, then these implementation details may help you determine how your search selector is likely to be used and how it should be implemented.

This function works with the implementation class `IlcMTMinimizeIntVarI` for constrained variables that are instances of `IlcIntVar`. The implementation class `IlcMTMinimizeIntVarI` is a subclass of `IlcMTSearchSelectorI`. The multithread class `IlcMTMinimizeIntVarI` differs from its sequential correspondent `IlcMinimizeIntVarI` in the following ways:

- `IlcMTMinimizeIntVarI` has an additional data member, `_vars` of type `IlcIntVarRef`. This data member stores a reference to all copies of the same variable to facilitate direct, simple access to any one of the copies.
- The constructor of `IlcMTMinimizeIntVarI` stores the variable with its name when the variable is constructed. When the variable is duplicated, the constructor reads the corresponding reference (`IlcIntVarRef`) in this way:

See Also: IlcMTSearchSelectorI

Global function operator>

```
public IloConstraint operator>(IloNumExprArg base, IloNumExprArg base2)
public IloConstraint operator>(IloNumExprArg base, IloNum val)
public IloConstraint operator>(IloNum val, IloNumExprArg eb)
public IloConstraint operator>(IloIntExprArg base, IloIntExprArg base2)
public IloConstraint operator>(IloIntExprArg base, IloInt val)
public IloConstraint operator>(IloInt val, IloIntExprArg eb)
```

Definition file: ilconcert/ilolinear.h

overloaded C++ operator
This overloaded C++ operator constrains its first argument to be strictly greater than its second argument. In order to be taken into account, this constraint must be added to a model and extracted for an algorithm.

Global function operator>

```
public IlcConstraint operator>(const IlcIntExp exp1, const IlcIntExp exp2)
public IlcConstraint operator>(const IlcIntExp exp1, IlcInt exp2)
public IlcConstraint operator>(IlcInt exp1, const IlcIntExp exp2)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This operator creates and returns an inequality constraint between its arguments (that is, the first must be strictly greater than the second).

When its arguments are constrained integer expressions, then when you post it, this constraint is associated with the `whenRange` propagation event.

When you create a constraint, it has no effect until you post it.

See Also: IlcConstraint, IlcIntExp, IlcLeOffset, operator<, operator<=, operator>=, operator!=, operator==

Global function operator>

```
public IloPredicate< IloObject > operator>(IloEvaluator< IloObject > left,
IloEvaluator< IloObject > right)
public IloPredicate< IloObject > operator>(IloEvaluator< IloObject > left, IloNum
threshold)
public IloPredicate< IloObject > operator>(IloNum threshold, IloEvaluator<
IloObject > right)
```

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

Creates a greater-than predicate from two evaluators.
These operators create a new `IloPredicate<IloObject>` instance by comparing the value returned by an evaluator with either that of another evaluator or a threshold value. The semantics of the new predicate are a greater-than comparison. The first function creates a predicate which returns `IloTrue` if and only if the value returned by the left evaluator is greater than the value returned by the right evaluator. The second function creates a predicate which returns `IloTrue` if and only if the value returned by the left evaluator is greater than the threshold value. The third function creates a predicate that returns `IloTrue` if and only if the threshold value is greater than the value returned by the right evaluator.

For more information, see Selectors.

Global function operator>

```
public IlcBool operator>(const IlcRevFloat & rev, IlcFloat value)
public IlcBool operator>(const IlcRevInt & rev, IlcInt value)
public IlcBool operator>(IlcInt value, const IlcRevInt & rev)
public IlcBool operator>(const IlcRevInt & rev1, const IlcRevInt & rev2)
public IlcBool operator>(IlcFloat value, const IlcRevFloat & rev)
public IlcBool operator>(const IlcRevFloat & rev1, const IlcRevFloat & rev2)
```

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>

This operator compares its arguments; if the first argument is strictly greater than the second, then it returns `IlcTrue`; otherwise, it returns `IlcFalse`.

See Also: IlcRevFloat, IlcRevInt

Global function IloUpdate

`public IloGoal **IloUpdate**(IloEnv env, IloMultipleEvaluator< IloObject, IloContainer > me, IloContainer container)`

Definition file: ilsolver/iimgoal.h
Include file: <ilsolver/iim.h>

A goal to update a multiple evaluator.
This goal is nothing more than a way to execute `IloMultipleEvaluator::update` in a goal, which can be particularly convenient when building goals using genetic algorithms.

The following code prepends a goal which makes sure a evaluator of the population is up-to-date before embarking on a new genetic algorithm generation.

```
generationGoal = IloUpdate(env, parentEvaluator, population)
              && generationGoal;
```

See Also: IloMultipleEvaluator

Global function IlcChooseFirstUnboundIntSet

`public IlcInt IlcChooseFirstUnboundIntSet(const IlcIntSetVarArray vars)`

Definition file: ilsolver/intset.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the first unbound constrained variable that it encounters in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseIntSetIndex, IloGenerate

Global function IloSample

```
public IloNHood IloSample(IloEnv env, IloNHood nhood, IloNum proportion, const char
* name=0)
public IloNHood IloSample(IloEnv env, IloNHood nhood, IloNum proportion, IloRandom
rand, const char * name=0)
```

Definition file: ilsolver/iimnhood.h
Include file: <ilsolver/iimls.h>

This function creates a neighborhood that samples a proportion `proportion` of neighborhood `nhood` at each move. Normally, the sampling is done in a round-robin fashion. For example, if the proportion was set to 0.4, then on the first move the neighborhood would correspond to the first 40% of the neighbors of `nhood`. On the next move, the neighborhood would be the next 40% of the neighbors of `nhood`. On the third move, the neighborhood would be the last 20% followed by the first 20% of the neighbors of `nhood`.

This round-robin behavior can be replaced with a completely random behavior by suppling the option argument `rand`. In this case, at each move (assuming `proportion`=0.4) 40% of the neighbors of `nhood` are sampled entirely at random by drawing random numbers from `rand`.

The optional argument `name`, if provided, becomes the name of the newly created neighborhood.

In such a neighborhood, the member function `IloNHood::define` defines the appropriate neighbor from `nhood` neighborhood using the sampling rule; the functions `IloNHood::notify` and `IloNHood#notifyOther` perform the corresponding actions using the same sampling rule; the function `IloNHood::start` calls `start` for `nhood`; and the function `IloNHood::getSize` returns `proportion` times `nhood.getSize()`, rounded to the nearest integer.

This type of neighborhood can be useful for simple diversification of search in a “best accept” context, where the best move is taken at each step of the search. By limiting the neighborhood at each step, it becomes possible to move out of local minima more easily. Such a neighborhood can also be useful in conjunction with tabu search.

See Also: IloNHood

Global function IlcAllMinDistance

```
public IlcConstraint IlcAllMinDistance(IlcIntVarArray vars, IlcInt k)
public IlcConstraint IlcAllMinDistance(IlcIntVarArray vars, IlcInt k,
IlcFilterLevel level)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

The function `IlcAllMinDistance` creates and returns a constraint. When that constraint is posted, it insures that the absolute distance between any pair of variables in the array `vars` will be greater than or equal to `k`. This function is for use during a Solver search, for example, inside a goal (an instance of `IlcGoal`) or inside a constraint (an instance of `IlcConstraint`). If you are looking for similar functionality for use in an Concert Technology *model*, consider `IloAllMinDistance` documented in the Concert API.

If you do not explicitly state a filter level, then Solver uses the default filter level for this constraint (that is, `IlcBasic`). The optional argument `level` can take one of the two values: `IlcBasic` or `IlcExtended`. The domain reduction during propagation depends on the value of `level`. See `IlcFilterLevel` for an explanation of filter levels and their effect on constraint propagation.

IlcBasic

`IlcBasic` is the lowest value.

IlcExtended

`IlcExtended` causes more domain reduction than `IlcBasic`; it also takes longer to run.

For more information, see `IloAllMinDistance`.

See Also: IlcAbs, IlcFilterLevel, IloAllMinDistance

Global function IloBestGenerate

```
public IloGoal IloBestGenerate(const IloEnv env, const IloNumVarArray vars, const
IloChooseIntIndex=IloChooseFirstUnboundInt)
public IloGoal IloBestGenerate(const IloEnv env, const IloAnyVarArray vars, const
IloChooseAnyIndex=IloChooseFirstUnboundAny)
public IloGoal IloBestGenerate(const IloEnv env, const IloAnyVarArray vars, const
IloChooseAnyIndex choose, const IloAnyValueSelector select)
public IloGoal IloBestGenerate(const IloEnv env, const IloNumVarArray vars, const
IloChooseIntIndex choose, const IloIntValueSelector select)
public IloGoal IloBestGenerate(const IloEnv env, const IloNumSetVarArray vars,
const IloChooseIntSetIndex=IloChooseFirstUnboundIntSet)
public IloGoal IloBestGenerate(const IloEnv env, const IloAnySetVarArray vars,
const IloChooseAnySetIndex=IloChooseFirstUnboundAnySet)
public IloGoal IloBestGenerate(const IloEnv env, const IloNumSetVarArray v, const
IloChooseIntSetIndex i, const IloIntSetValueSelector s)
public IloGoal IloBestGenerate(const IloEnv env, const IloAnySetVarArray v, const
IloChooseAnySetIndex i, const IloAnySetValueSelector s)
```

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal, using the criterion indicated by `choose` and the search selector indicated by `select`. This goal is part of the enumeration algorithm available in an instance of `IloSolver`. It enables you to set parameters for choosing the order in which variables are tried during the search for a solution.

This goal binds each constrained variable in its argument `vars`. It does so by calling the function `IloBestInstantiate` for each of them. You control the order in which the variables are bound by means of the criterion `choose`. The argument select is passed to each call to `IloBestInstantiate`, if that argument is provided.

The goal returned by this function differs from the goal returned by `IloGenerate`: this one calls `IloBestInstantiate`, which tries only one value for each variable, whereas `IloGenerate` calls `IloInstantiate`, which may try all values in the domain of each variable.

When this function takes an instance of the class `IloEnv` as a parameter, it returns an instance of `IloGoal` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the goal that it returns as an instance of `IlcGoal` for use during a Solver search.

This function works on numerical variables of type `Float` and type `Int`.

See Also: IloBestInstantiate, IloGenerate, IloGoal, IloInstantiate, IlcBestGenerate

Global function IlcUnion

```
public IlcIntSetVar IlcUnion(IlcIntSetVar set, IlcIntToIntExpFunction F)
public IlcIntSetVar IlcUnion(IlcAnySetVar set, IlcAnyToIntExpFunction F)
public IlcAnySetVar IlcUnion(IlcIntSetVar set, IlcIntToAnyExpFunction F)
public IlcAnySetVar IlcUnion(IlcAnySetVar set, IlcAnyToAnyExpFunction F)
```

Definition file: ilsolver/setcst.h
Include file: <ilsolver/ilosolver.h>

These functions create and return a new set variable that represents the union of the values returned by the function `F` when applied to the elements of the constrained set variable `set`. The values returned by `F` are constrained expressions or variables (that is, instances of `IlcIntExp`, `IlcIntVar`, `IlcAnyExp`, or `IlcAnyVar`).

Adding These Constrained Variables

You may add these constrained variables only during a Solver search; that is, inside a goal (an instance of `IlcGoal`) or inside a constraint (an instance of `IlcConstraint`). If you are looking for similar functionality in a constraint to add to a model, see `IloEqUnion`.

Example

These `IlcUnion` functions can be useful to express constraints on the values of a constrained attribute of a computed set of objects.

For example, if we have to connect cards to a rack, and if for each card we also have to connect one sensor, it is possible to express constraints on the set of sensors that are connected to the cards in the rack.

```
class Sensor {
public:
   const char* _name;
   Sensor(IloSolver s, const char* name);
};
class Card {
public:
   IlcAnyVar _sensor;
   Card(IloSolver s, IlcAnyArray sensors)
    :_sensor(s, sensors) {}
};
//Access to the sensor connected to a card
IlcAnyToAnyExpFunction sensorsAccess;
//The possible sensors
IlcAnyArray sensors(s, 4);
sensors[0] = new (s.getHeap()) Sensor(s, "Sensor#0");
sensors[1] = new (s.getHeap()) Sensor(s, "Sensor#1");
sensors[2] = new (s.getHeap()) Sensor(s, "Sensor#2");
sensors[3] = new (s.getHeap()) Sensor(s, "Sensor#3");
//The possible cards
IlcAnyArray cards(m, 3);
cards[0] = new (s.getHeap()) Card(s, sensors);
cards[1] = new (s.getHeap()) Card(s, sensors);
cards[2] = new (s.getHeap()) Card(s, sensors);
//The cards connected to the rack
IlcAnySetVar rackCards(s, cards, "Rack#1");
//The sensors connected to the rack
IlcAnySetVar rackSensors = IlcUnion(rackCards, sensorAccess);
```

Here is how we add these constrained variables inside a constraint:

```
//At most 10 sensors in the rack
s.add(IlcCard(rackSensors) <= 10);
//sensor#1 in a card of the rack
s.add(IlcMember(sensors[1], rackSensors));
```

Of course, it is possible to compose several levels of indirection. For example, we can post constraints on the processes assigned to the sensors which are connected to the cards that are plugged into a rack:

```
IlcUnion(IlcUnion(rackCards, sensorAccess), processAccess);
```

See Also: IlcAnySetVar, IlcEqUnion, IlcIntSetVar, IloEqUnion

Global function IlcUnion

```
public IlcIntSetVar IlcUnion(IlcIntSetVar index, IlcIntToIntSetVarFunction f)
public IlcIntSetVar IlcUnion(IlcAnySetVar set, IlcAnyToIntSetVarFunction F)
public IlcAnySetVar IlcUnion(IlcIntSetVar set, IlcIntToAnySetVarFunction F)
public IlcAnySetVar IlcUnion(IlcAnySetVar set, IlcAnyToAnySetVarFunction F)
```

Definition file: ilsolver/setcst.h
Include file: <ilsolver/ilosolver.h>

These functions create and return a new set variable that represents the union of the values returned by the function `F` when applied to the elements of the constrained set variable `set`. The values returned by `F` are constrained set variables (`IlcIntSetVar`, `IlcAnySetVar`). These `IlcUnion` functions can be useful to express constraints on the values of a constrained set attribute of a computed set of objects.

Adding These Constrained Variables

You may add these constrained variables only during a Solver search; that is, inside a goal (an instance of `IlcGoal`) or inside a constraint (an instance of `IlcConstraint`). If you are looking for similar functionality in a constraint to add to a *model*, see `IloEqUnion`.

Example

For example, if we have to connect cards to a rack, and if for each card we also have to connect a set of sensors, it is possible to express constraints on the set of sensors that are connected to the cards in the rack.

```
class Sensor {
public:
   const char* _name;
   Sensor(IloSolver s, const char* name);
};
class Card {
public:
   IlcAnySetVar _sensors;
   Card(IloSolver s, IlcAnyArray sensors)
    :_sensors(s, sensors) {}
};
//Access to the sensor connected to a card
IlcAnyToAnySetVarFunction sensorsAccess;
//The possible sensors
IlcAnyArray sensors(s, 4);
sensors[0] = new (s.getHeap()) Sensor(s, "Sensor#0");
sensors[1] = new (s.getHeap()) Sensor(s, "Sensor#1");
sensors[2] = new (s.getHeap()) Sensor(s, "Sensor#2");
sensors[3] = new (s.getHeap()) Sensor(s, "Sensor#3");
//The possible cards
IlcAnyArray cards(s, 3);
cards[0] = new (s.getHeap()) Card(s, sensors);
cards[1] = new (s.getHeap()) Card(s, sensors);
cards[2] = new (s.getHeap()) Card(s, sensors);
//The cards connected to the rack
IlcAnySetVar rackCards(s, cards, "Rack#1");
//The sensors connected to the rack
IlcAnySetVar rackSensors = IlcUnion(rackCards, sensorAccess);
```

Here is how we add these constrained variables *inside* a constraint:

```
//At most 10 sensors in the rack
s.add(IlcCard(rackSensors) <= 10);
//sensor#1 in a card of the rack
s.add(IlcMember(sensors[1], rackSensors));
```

Of course, it is possible to compose several levels of indirection. For example, we can access the processes assigned to the sensors which are connected to the cards that are plugged into a rack:

```
IlcUnion(IlcUnion(rackCards, sensorAccess), processAccess);
```

See Also: IloEqUnion, IlcAnySetVar, IlcEqUnion, IlcIntSetVar

Global function IlcUnion

```
public IlcIntSetVar IlcUnion(IlcIntSetVar set, IlcIntToIntFunction F)
public IlcIntSetVar IlcUnion(IlcAnySetVar set, IlcAnyToIntFunction F)
public IlcAnySetVar IlcUnion(IlcIntSetVar set, IlcIntToAnyFunction F)
public IlcAnySetVar IlcUnion(IlcAnySetVar set, IlcAnyToAnyFunction F)
```

Definition file: ilsolver/setcst.h
Include file: <ilsolver/ilosolver.h>

These functions create and return a new set variable that represents the union of the values returned by the function `F` when applied to the elements of the constrained set variable `set`. These functions can be useful to get and constrain the values of an attribute of a computed set of objects.

Adding These Constrained Variables

You may add these constrained variables only during a Solver search; that is, inside a goal (an instance of `IlcGoal`) or inside a constraint (an instance of `IlcConstraint`). If you are looking for similar functionality in a constraint to add to a *model*, see `IloEqUnion` documented in the *ILOG Concert Technology Reference Manual*.

Example

For example, if we have to assign crew members to a flight and if each crew member has an attribute that describes the language he or she speaks, with this `IlcUnion` expression, it is possible to post constraints on the set of languages that must be spoken for the flight, like this:

```
enum Language {English, French, German};
class CrewMember {
public:
   const char* _name;
   IlcInt      _language;
   CrewMember(IloSolver s, const char* name, Language lang);
};
//Access to the language spoken by a crew member
IlcAnyToIntFunction languages;
//Possible crew members
IlcAnyArray c(s, 3);
c[0]= new (s.getHeap()) CrewMember(s, "John",  English);
c[1]= new (s.getHeap()) CrewMember(s, "Kai",   German);
c[2]= new (s.getHeap()) CrewMember(s, "Julie", French);
//The flight
IlcAnySetVar crew(s, c, "NewYork-Paris");
//The languages spoken on this flight
IlcIntSetVar langs = IlcUnion(crew, languages);
//At least 2 different languages spoken
s.add(IlcCard(langs) >= 2);
//French must be spoken
s.add(IlcMember(French, langs));
```

See Also: IlcAnySetVar, IlcEqUnion, IlcIntSetVar

Global function IlcUnion

```
public IlcIntSetVar IlcUnion(IlcIntSetVar var1, IlcIntSetVar var2)
public IlcAnySetVar IlcUnion(IlcAnySetVar var1, IlcAnySetVar var2)
public IlcIntSetVar IlcUnion(IlcIntSetVarArray vars)
public IlcAnySetVar IlcUnion(IlcAnySetVarArray vars)
```

Definition file: ilsolver/intset.h
Include file: <ilsolver/ilosolver.h>

These functions:

- Return the union of `var1` and `var2`. The parameters `var1` and `var2` must be built on the same initial array.
- Create and return a new set variable that represents the union of the variables `vars`. All the variables in `vars` must be built on the same initial array.

Adding These Constrained Variables

You may add these constrained variables only during a Solver search; that is, inside a goal (an instance of `IlcGoal`) or inside a constraint (an instance of `IlcConstraint`). If you are looking for similar functionality in a constraint to add to a *model*, see `IloEqUnion` documented in the *Concert Technology Reference Manual*.

See Also: IlcAnySetVar, IlcAnySetVarArray, IlcCard, IlcEqUnion, IlcIntSetVar, IlcIntSetVarArray

Global function IloStoreBestSolution

```
public IlcGoal IloStoreBestSolution(IloSolver solver, IloSolution solution)
public IloGoal IloStoreBestSolution(IloEnv env, IloSolution solution)
```

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function returns a goal which stores solution using `solution.store(solver)`, if the value of the objective of solution is worse than the currently instantiated value. What is considered “worse” depends on the sense of the objective added to solution.

For more information, see `IloSolution`.

See Also: IloRestoreSolution, IloSolution, IloStoreSolution, IloUpdateBestSolution

Global function IloFlip

```
public IloNHood IloFlip(IloEnv env, IloIntVarArray vars, const char * name=0)
```

Definition file: ilsolver/iimnhood.h
Include file: <ilsolver/iimls.h>

This function returns a neighborhood that can be used to “flip” variables in a local search problem involving binary variables (those with a value of 0 or 1).

The function defines a neighborhood that flips the value of a single variable in `vars`. Specifically, for each variable in `vars`, there is a neighbor in the neighborhood that flips its value (a change from 0 to 1, or from 1 to 0). The optional argument `name`, if supplied, becomes the name of the returned neighborhood.

See Also: IloNHood, IloNHoodI

Global function IlcSizeOverDegreeVarEvaluator

```
public IloEvaluator< IlcIntVar > IlcSizeOverDegreeVarEvaluator(IloEnv env)
public IloEvaluator< IlcIntVar > IlcSizeOverDegreeVarEvaluator(IloSolver solver)
```

Definition file: ilsolver/custgoal.h
Include file: <ilsolver/ilosolver.h>

This function returns an instance of the evaluator `IloEvaluator<IlcIntVar>`. The evaluation returns the floating-point value `IlcFloat(x.getSize()) / solver.getDegree(x)`, where `x` is the evaluated variable.

Global function IloGoalTrue

`public IloGoal IloGoalTrue(const IloEnv env)`

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal that always succeeds. It is one of the predefined building blocks (or search primitives) in an IBM® ILOG® Solver search.

This function returns an instance of `IloGoal` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the goal that it returns as an instance of `IlcGoal` for use during a Solver search.

See Also: IloGoal

Global function IloMonotonicDecreasingNumExpr

`public IloNumExprArg` **`IloMonotonicDecreasingNumExpr`**`(IloNumExprArg node, IloNumFunction f, IloNumFunction invf)`

Definition file: ilconcert/iloexpression.h

For constraint programming: creates a new constrained expression equal to `f(x)`.
This function creates a new constrained expression equal to `f(x)`. The arguments `f` and `invf` must be pointers to functions of type `IloNumFunction`. Those two functions must be inverses of one another, that is,

`invf(f(x)) == x` and `f(invf(x)) == x` for all `x`.

Those two functions must also be monotonically decreasing.

`IloMonotonicDecreasingNumExpr` does *not* verify whether `f` and `invf` are inverses of one another. It does *not* verify whether they are monotonically decreasing either.

The effects of this function are reversible.

Global function IloChooseMaxMinInt

`public IlcInt` **`IloChooseMaxMinInt`**`(const IlcIntVarArray vars)`

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the greatest minimum from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseIntIndex, IloGenerate

Global function ilc_trace_stop_here

`public void` **`ilc_trace_stop_here`**`()`

Definition file: ilsolver/ilcerr.h
Include file: <ilsolver/ilosolver.h>

The function `ilc_trace_stop_here` is useful when you are running a Solver program from a debugger. With it, you can execute the constraint propagation step by step. A breakpoint can be set for the function `ilc_trace_stop_here` to stop the program. You do that in this way if you are using `dbx`, for example:

```
(dbx) stop in ilc_trace_stop_here
```

On some platforms, such as Windows NT, for example, you must prefix an underscore to the name of the function, like this: `_ilc_trace_stop_here`.

See Also: IlcTrace, IlcTraceI

Global function IlcSubsetEq

```
public IlcConstraint IlcSubsetEq(IlcAnySetVar a, IlcAnySetVar b)
public IlcConstraint IlcSubsetEq(IlcAnySet a, IlcAnySetVar b)
public IlcConstraint IlcSubsetEq(IlcAnySetVar a, IlcAnySet b)
public IlcConstraint IlcSubsetEq(IlcIntSetVar a, IlcIntSetVar b)
public IlcConstraint IlcSubsetEq(IlcIntSet a, IlcIntSetVar b)
public IlcConstraint IlcSubsetEq(IlcIntSetVar a, IlcIntSet b)
```

Definition file: ilsolver/ilcset.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a constraint that `a` must be a subset of `b`. (The set `a` may be equal to the set `b`.) To set a strict subset constraint, use the following statement:

```
s.add(IlcSubset(set1, set2));
```

Adding These Constraints

You may add these constraints only during a Solver search; that is, inside a goal (an instance of `IlcGoal`) or inside a constraint (an instance of `IlcConstraint`). If you are looking for similar functionality in a constraint to add to a model, see `IloEqUnion`.

See Also: IlcAnySetVar, IlcConstraint, IlcIntSetVar, IlcSubset, IloEqUnion

Global function IlcMonotonicDecreasingFloatExp

```
public IlcFloatExp IlcMonotonicDecreasingFloatExp(IlcFloatExp x, IlcFloatFunction
f, IlcFloatFunction invf, const char * functionName=0)
```

Definition file: ilsolver/nlinflt.h
Include file: <ilsolver/ilosolver.h>

This function creates a new constrained expression equal to `f(x)`. The arguments `f` and `invf` must be pointers to functions of type `IlcFloatFunction`. Those two functions must be inverses of one another, that is,

`invf(f(x)) == x` and `f(invf(x)) == x` for all `x`.

Those two functions must also be monotonically decreasing.

`IlcMonotonicDecreasingFloatExp` does *not* verify whether `f` and `invf` are inverses of one another. It does *not* verify whether they are monotonically decreasing either.

The effects of this function are reversible.

See Also: IlcExponent, IlcFloatExp, IlcFloatFunction, IlcLog, IlcMonotonicIncreasingFloatExp, IlcPower, IlcSquare

Global function IlcGoalFail

`public IlcGoal **IlcGoalFail**(IloSolver solver, IlcAny label=0)`

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>

The function `IlcGoalFail` calls `IloSolver::fail(label)` in its body. Thus it always fails.

Example

This function can be used to search for all possible ways to execute a goal. The idea is to try all possible subgoals of all choice points by calling `IlcGoalFail` after each successful goal execution. Then Solver backtracks and searches for another execution.

Thus if the statements

```
solver.solve(IlcGoal);
```

search for one successful execution of the goal, `goal`, then the statement

```
solver.solve(IlcAnd(goal, IlcGoalFail(solver)));
```

search for all possible executions of `goal`.

See Also: IlcOr, ILCGOAL0, IlcGoal, IloSolver, IlcAnd

Global function operator&&

```
public IlcConstraint operator&&(const IlcConstraint ct1, const IlcConstraint ct2)
```

Definition file: ilsolver/numi.h
Include file: <ilsolver/ilosolver.h>

This operator creates and returns a constraint: the conjunction of its arguments.

When you create a constraint, it has no effect until you post it.

See Also: IlcConstraint

Global function operator&&

`public IloAnd operator&&(const IloConstraint constraint1, const IloConstraint constraint2)`

Definition file: ilconcert/ilomodel.h

Overloaded C++ operator for conjunctive constraints.
This overloaded C++ operator creates a conjunctive constraint that represents the conjunction of its two arguments. The constraint can represent a conjunction of two constraints; of a constraint and another conjunction; or of two conjunctions. In order to be taken into account, this constraint must be added to a model and extracted by an algorithm, such as `IloCplex` or `IloSolver`.

Global function operator&&

```
public IloPoolOperator operator&&(IloPoolOperator op1, IloPoolOperator op2)
```

Definition file: ilsolver/iimoperator.h
Include file: <ilsolver/iim.h>

Produces the conjunction of two operators.
This operator produces a conjunction operator which, when invoked, invokes `op1`. If successful, it invokes `op2`. If `op2` then succeeds, the conjunction is deemed to have succeeded.

Note
If no prototype is set on the conjunction, then its prototype is defined to be one from either `op1` or `op2`. If both suboperators have prototypes defined, that of `op2` is chosen.

Global function operator&&

`public IloPredicate< IloObject > operator&&(IloPredicate< IloObject > left, IloPredicate< IloObject > right)`

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

Creates a predicate performing AND on two predicates.
This operator creates a new `IloPredicate<IloObject>` instance from two `IloPredicate<IloObject>` instances. The semantics of the combination of the component predicates is that of logical `AND`. That is, the combined predicate will return `IloTrue` for a particular object if and only if both the component predicates return `IloTrue` for that object.

For more information, see Selectors.

Global function IlcArcCos

```
public IlcFloatExp IlcArcCos(const IlcFloatExp x)
public IlcFloat IlcArcCos(IlcFloat x)
```

Definition file: ilsolver/nlinflt.h
Include file: <ilsolver/ilosolver.h>

When its argument is a constrained floating-point expression, this function creates a constrained floating-point expression (that is, an instance of `IlcFloatExp` or one of its subclasses) which is equal to the arc cosine (in the range 0 to Pi) of its argument `x` expressed in radians. The effects of this function are reversible.

When its argument is an unconstrained numeric value (that is, a value of type `IlcFloat`), this function returns the arc cosine of its argument.

If you want to manipulate constrained floating-point expressions in degrees, we strongly recommend that you call the trigonometric functions on variables expressed in radians and then convert the results to degrees (rather than declaring the constrained floating-point expressions in degrees and then converting them to radians to call the trigonometric functions).

The reason for that advice is that the method we recommend gives more accurate results in the context of the usual floating-point pitfalls. See the *IBM ILOG Solver User's Manual* for an explanation of those pitfalls.

See Also: IlcArcSin, IlcArcTan, IlcCos, IlcDegToRad, IlcFloatExp, IlcHalfPi, IlcQuarterPi, IlcPi, IlcRadToDeg, IlcSin, IlcTan, IlcThreeHalfPi, IlcTwoPi

Global function IlcDegToRad

```
public IlcFloat IlcDegToRad(IlcFloat angle)
```

Definition file: ilsolver/nlinflt.h

This function converts an angle expressed in degrees to an angle expressed in radians.

If you want to manipulate constrained floating-point expressions in degrees, we strongly recommend that you call the trigonometric functions on variables expressed in radians and then convert the results to degrees (rather than declaring the constrained floating-point expressions in degrees and then converting them to radians to call the trigonometric functions).

The reason for that advice is that the method we recommend gives more accurate results in the context of the usual floating-point pitfalls. See the *IBM ILOG Solver User's Manual* for an explanation of those pitfalls.

Good practice:

```
IloSolver s;
IlcFloatVar x(s,0,IlcPi);
IlcFloatExp y=IlcSin(x);
// ... constraints on x and y ...
```

Global function IlcGoalTrue

```
public IlcGoal IlcGoalTrue(IloSolver solver)
```

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal that always succeeds. It is one of the predefined building blocks (or search primitives) in an IBM® ILOG® Solver search.

This function returns an instance of `IlcGoal` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`.

See Also: IlcGoal

Global function IloMin

`public IloNumToNumStepFunction **IloMin**(const IloNumToNumStepFunction f1, const IloNumToNumStepFunction f2)`

Definition file: ilconcert/ilonumfunc.h

Creates and returns a function equal to the minimal value of its argument functions.
This operator creates and returns a function equal to the minimal value of the functions `f1` and `f2`. That is, for all points `x` in the definition interval, the resulting function is equal to the `min(f1(x), f2(x))`. The argument functions `f1` and `f2` must be defined on the same interval. The resulting function is defined on the same interval as the arguments. See also: `IloNumToNumStepFunction`.

Global function IloMin

```
public IloNum IloMin(const IloNumArray vals)
public IloNum IloMin(IloNum val1, IloNum val2)
public IloInt IloMin(const IloIntArray vals)
public IloNumExprArg IloMin(const IloNumExprArray exprs)
public IloIntExprArg IloMin(const IloIntExprArray exprs)
public IloNumExprArg IloMin(const IloNumExprArg x, const IloNumExprArg y)
public IloNumExprArg IloMin(const IloNumExprArg x, IloNum y)
public IloNumExprArg IloMin(IloNum x, const IloNumExprArg y)
public IloIntExprArg IloMin(const IloIntExprArg x, const IloIntExprArg y)
public IloIntExprArg IloMin(const IloIntExprArg x, IloInt y)
public IloNumExprArg IloMin(const IloIntExprArg x, IloNum y)
public IloIntExprArg IloMin(const IloIntExprArg x, int y)
public IloIntExprArg IloMin(IloInt x, const IloIntExprArg y)
public IloNumExprArg IloMin(IloNum x, const IloIntExprArg y)
public IloIntExprArg IloMin(int x, const IloIntExprArg y)
```

Definition file: ilconcert/iloexpression.h

Returns a numeric value representing the min of numeric values.
These functions compare their arguments and return the least value. When its argument is an array, the function compares the *elements* of that array and returns the least value.

Global function IloMin

```
public IloEvaluator< IloObject > IloMin(IloEvaluator< IloObject > left,
IloEvaluator< IloObject > right)
public IloEvaluator< IloObject > IloMin(IloEvaluator< IloObject > left, IloNum c)
public IloEvaluator< IloObject > IloMin(IloNum c, IloEvaluator< IloObject > right)
```

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

These functions create a composite `IloEvaluator<IloObject>` instance. These evaluators return the least value of the float values returned by the two evaluators given as argument, or the least value between the float value and the evaluator given as argument.

For more information, see Selectors.

Global function IlcSolveBounds

```
public void IlcSolveBounds(IlcFloatVar var, IlcFloat prec=.1)
public void IlcSolveBounds(IlcFloatVarArray array, IlcFloat prec=.1)
```

Definition file: ilsolver/nlinflt.h
Include file: <ilsolver/ilosolver.h>

The function `IlcSolveBounds` efficiently reduces the domain of a constrained floating-point expression, regardless of whether the expression is written in canonical or factored form. One version of this function works on a constrained floating-point variable (that is, an instance of `IlcFloatVar`); the other works on an array of constrained floating-point variables (that is, an instance of `IlcFloatVarArray`).

This function reduces the domain of its argument by propagating constraints more than usual. It checks whether the boundaries of the domain of its argument are consistent with all the constraints posted on its argument. If that is not the case, the interval will be reduced until the boundaries become consistent up to the relative precision indicated by the argument `prec`. That argument can be used to control the execution of this function. If `prec` is small, the new domain computed by `IlcSolveBounds` will be smaller. However, the smaller `prec`, the longer `IlcSolveBounds` will take.

For more information, see the concept Propagation.

See Also: IlcFloatVar, IlcFloatVarArray

Global function IlcEqIntersection

```
public IlcConstraint IlcEqIntersection(IlcIntSetVar intersection, IlcIntSetVar
var1, IlcIntSetVar var2)
public IlcConstraint IlcEqIntersection(IlcAnySetVar intersection, IlcAnySetVar
var1, IlcAnySetVar var2)
public IlcConstraint IlcEqIntersection(IlcIntSetVar intersection, IlcIntSetVarArray
vars)
public IlcConstraint IlcEqIntersection(IlcAnySetVar intersection, IlcAnySetVarArray
vars)
```

Definition file: ilsolver/intset.h
Include file: <ilsolver/ilosolver.h>

These functions:

- create and return a constraint that forces the intersection of `var1` and `var2` to be equal to the value of `intersection;`
- create and return a constraint that forces the intersection of the `vars` variables to be equal to the value of `intersection`. The variable `intersection` and all the variables in `vars` must be built on the same initial array.

See Also: IlcAnySetVar, IlcAnySetVarArray, IlcConstraint, IlcIntersection, IlcIntSetVar, IlcIntSetVarArray

Global function IloPartition

```
public IloConstraint IloPartition(const IloEnv env, const IloAnySetVarArray vars)
public IloConstraint IloPartition(const IloEnv env, const IloAnySetVarArray vars,
const IloAnyArray vals)
public IloConstraint IloPartition(const IloEnv, const IloIntSetVarArray vars)
public IloConstraint IloPartition(const IloEnv, const IloIntSetVarArray vars, const
IloIntArray vals)
```

Definition file: ilconcert/iloanyset.h

For IBM® ILOG® Solver: a constraint forcing each value of an array to be required by one set variable in an array.
This function creates and returns a constraint. When that constraint is posted, it insures that each value of the array `vals` will be required by exactly one set variable of the array `vars`.

If the argument `vals` is not mentioned, the array `vals` is formed by the union of the values involved in the set variables of `vars`.

In this context, a constraint will be posted after it has been added to a model and extracted by a solver (for example, an instance of `IloSolver` documented in the *IBM ILOG Solver Reference Manual*).

Global function IlcPartition

```
public IlcConstraint IlcPartition(IlcIntSetVarArray vars, IlcIntArray val)
public IlcConstraint IlcPartition(IlcAnySetVarArray vars, IlcAnyArray val,
IlcFilterLevel level)
public IlcConstraint IlcPartition(IlcAnySetVarArray vars, IlcAnyArray val)
public IlcConstraint IlcPartition(IlcIntSetVarArray vars, IlcIntArray val,
IlcFilterLevel level)
public IlcConstraint IlcPartition(IlcAnySetVarArray vars, IlcFilterLevel level)
public IlcConstraint IlcPartition(IlcAnySetVarArray vars)
public IlcConstraint IlcPartition(IlcIntSetVarArray vars, IlcFilterLevel level)
public IlcConstraint IlcPartition(IlcIntSetVarArray vars)
```

Definition file: ilsolver/ilcset.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a constraint. When that constraint is posted, it insures that each value of the array `val` will be required by exactly one set variable of the array `vars`.

If the argument `val` is not mentioned, the array `val` is formed by the union of the values involved in the set variables of `vars`.

If you do not explicitly state a filter level, then Solver uses the default filter level for this type of constraint. The optional argument `level` can take one of the two values: `IlcBasic`, `IlcExtended`. The domain reduction during propagation depends on the value of `level`.

`IlcBasic` is the lowest value.

`IlcExtended` causes more domain reduction than `IlcBasic`; it also takes longer to run.

See `IlcFilterLevel` for an explanation of filter levels and their effect on constraint propagation.

See Also: IlcFilterLevel, IlcAllNullIntersect

Global function IlcLogicAggregator

`public IlcConstraintAggregator IlcLogicAggregator(IloSolver solver)`

Definition file: ilsolver/ilosolverhandle.h
Include file: <ilsolver/ilosolver.h>

This aggregator groups logical constraints of the form:

```
IloIfThen(ct1, ct1)
```

and

```
ct1 == ct2
```

where `ct1` and `ct2` are constraints of the type `x == a` or `x != a` (where `x` is an `IloIntVar` and `a` an integer constant). It improves propagation efficiency and reduces memory consumption on these constraints.

Global function IlcBFSEvaluator

```
public IlcNodeEvaluator IlcBFSEvaluator(IloSolver solver, IlcIntVar var, IlcInt
step=1)
public IlcNodeEvaluator IlcBFSEvaluator(IloSolver solver, IlcFloatVar var, IlcFloat
step=1.0)
```

Definition file: ilsolver/search.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a *node evaluator* that implements a best first search.

Nodes are evaluated according to the variable `var`. As long as the minimum of `var` is no greater than the minimum of the evaluation of each open nodes `+ step`, the search continues as a depth-first search. If the opposite is true, the goal manager postpones the evaluation of the current node and jumps to the best open node.

See Also: IlcNodeEvaluator

Global function IloFailLimit

```
public IloSearchLimit IloFailLimit(const IloEnv env, IloInt maxNbFails)
```

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a search limit that limits the exploration of the search tree during the search for a solution by stopping the search when a given number of failures (`maxNbFails`) have occurred. A fail limit is useful in a goal, such as the one returned by the function `IloLimitSearch` or other instances of `IloGoal`, to control the exploration of the search tree.

When this function takes an instance of the class `IloEnv` as a parameter, it returns an instance of `IloSearchLimit` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the search limit that it returns as an instance of `IlcSearchLimit` for use during a Solver search.

See Also: IloGoal, IloLimitSearch, IloSearchLimit, IlcSearchLimit

Global function IloAndGoal

`public IloGoal IloAndGoal(const IloEnv, const IloGoal, const IloGoal)`

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal for use in a search. This goal represents the conjunction (logical AND) of its two goal arguments. You can replace the code `IloAndGoal(env, g1, g2)` with `g1 && g2`.

When it takes an instance of the class `IloEnv` as a parameter, it returns an instance of `IloGoal` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the goal that it returns as an instance of `IlcGoal` for use during a Solver search.

See Also: IlcGoal, IloGoal, IloSolver, IlcAnd

Global function IlcIntersection

```
public IlcIntSetVar IlcIntersection(IlcIntSetVar var1, IlcIntSetVar var2)
public IlcAnySetVar IlcIntersection(IlcAnySetVar var1, IlcAnySetVar var2)
public IlcIntSetVar IlcIntersection(IlcIntSetVarArray vars)
public IlcAnySetVar IlcIntersection(IlcAnySetVarArray vars)
```

Definition file: ilsolver/intset.h
Include file: <ilsolver/ilosolver.h>

These functions:

- return the intersection of `var1` and `var2`;
- create and return a new set variable that represents the intersection of the `vars` variables. All the variables in `vars` must be built on the same initial array.

See Also: IlcAnySetVar, IlcAnySetVarArray, IlcEqIntersection, IlcIntSetVar, IlcIntSetVarArray

Global function IlcChooseMaxRegretMin

`public IlcInt **IlcChooseMaxRegretMin**(const IlcIntVarArray vars)`

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the maximal difference between the minimal possible value and the next minimal possible value from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is based on the principle of *least regret*. Regret is the difference between what would have been the best possible decision in a scenario and what was the actual decision.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseIntIndex, IloGenerate

Global function IloEnableNANDetection

```
public void IloEnableNANDetection()
```

Definition file: ilconcert/ilosys.h

Enables NaN (Not a number) detection.
This function enables your application to detect invalid data as a NaN (Not a number).

Global function IloSubsetEq

```
public IloConstraint IloSubsetEq(const IloEnv env, const IloAnySetVar var1, const
IloAnySetVar var2)
public IloConstraint IloSubsetEq(const IloEnv env, const IloAnySet var1, const
IloAnySetVar var2)
public IloConstraint IloSubsetEq(const IloEnv env, const IloAnySetVar var1, const
IloAnySet var2)
public IloConstraint IloSubsetEq(const IloEnv, const IloIntSetVar var1, const
IloIntSetVar var2)
public IloConstraint IloSubsetEq(const IloEnv, const IloIntSet var1, const
IloIntSetVar var2)
public IloConstraint IloSubsetEq(const IloEnv, const IloIntSetVar var1, const
IloIntSet var2)
```

Definition file: ilconcert/iloanyset.h

For IBM® ILOG® Solver: a constraint forcing one set to be a subset of or equivalent to another set.
This function creates and returns a constraint (an instance of `IloConstraint`) for use in a model. That constraint forces `var1` to be a subset of `var2`. (The set `var1` may be equal to `var2`; every element of `var1` is also an element of `var2`.)

In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

Global function IloIDFSEvaluator

`public IloNodeEvaluator **IloIDFSEvaluator**(const IloEnv env, IloInt max)`

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a *node evaluator* that implements limited interleaved depth-first search in a Concert Technology model. (The function `IloDFSEvaluator` implements conventional depth-first search.)

The parameter `max` is the maximum depth where this evaluator is active. Below this depth, it acts as a depth-first search evaluator.

This function returns an instance of `IloNodeEvaluator` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the node evaluator that it returns as an instance of `IlcNodeEvaluator` for use during a Solver search.

See Also: IloDFSEvaluator, IlcNodeEvaluator, IloNodeEvaluator

Global function IloScanDeltas

```
public IloGoal IloScanDeltas(IloEnv env, IloSolution solution, IloSolutionArray
deltas)
public IloGoal IloScanDeltas(IloEnv env, IloSolution solution, IloSolutionArray
deltas, IloNeighborIdentifier nid)
public IlcGoal IloScanDeltas(IloSolver solver, IloSolution solution,
IloSolutionArray deltas)
public IlcGoal IloScanDeltas(IloSolver solver, IloSolution solution,
IloSolutionArray deltas, IlcNeighborIdentifier nid)
```

Definition file: ilsolver/iimls.h
Include file: <ilsolver/iimls.h>

These goals scan the array of deltas `deltas`. These goals test the set of deltas using `solution` as the current solution reference. Each leaf node of the above goals corresponds to the application of a different delta from `deltas` to the solution `solution`.

If specified, `nid` is used to hold the index (in the array of deltas) of the deltas, and the delta itself instantiated at the current leaf. This can be used to communicate with other goals such as `IloNotify` and `IloTest`.

If any applied delta is to be saved back to `solution`, this must be done explicitly via `solution.store(solver)`, or using `IloStoreSolution`.

For more information, see `IloSolution` in the *Concert Technology Reference Manual*.

See Also: IloApplyMetaHeuristic, IloNotify, IloScanNHood, IloStoreSolution, IloTest

Global function IloSBSEvaluator

```
public IloNodeEvaluator IloSBSEvaluator(const IloEnv env, IloInt step=4, IloInt
maxDiscrepancy=IloIntMax)
```

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a node evaluator (an instance of `IloNodeEvaluator`) that implements Slice-Based Search in a Concert Technology model.

A *discrepancy* is a right move in the path from the root of the search tree to the current node. The intuition is that a left move (the first goal in an `IlcOr` goal) is better than a right move (the second goal). Thus, by limiting the number of discrepancies, we try to stick close to the search heuristics (the goals to search for a solution of the problem being solved).

This implementation of Slice-Based Search divides the search tree into slices. Slice `k` corresponds to the open nodes of the search tree with a number of discrepancies between `k*step` and `(k + 1)*step - 1`. The node evaluator indicates that the search should explore slice `0`, then slice `1`, then slice `2`, and so on.

This evaluator also discards nodes with a number of discrepancies greater than `maxDiscrepancy`. This technique is appropriate with constraint programming and generally offers better performance than depth-first search.

This function returns an instance of `IloNodeEvaluator` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the node evaluator that it returns as an instance of `IlcNodeEvaluator` for use during a Solver search.

See Also: IloApply, IloNodeEvaluator, IlcNodeEvaluator

Global function IloRound

`public IloNum **IloRound**(IloNum val)`

Definition file: ilconcert/iloenv.h

Computes the nearest integer value to its argument.
This function computes the nearest integer value to `val`. Halfway cases are rounded to the larger in magnitude.

Examples:

```
IloRound(IloInfinity) is IloInfinity.
IloRound(-IloInfinity) is -IloInfinity.
IloRound(0) is 0.
IloRound(0.4) is 0.
IloRound(-0.4) is 0.
IloRound(0.5) is 1.
IloRound(-0.5) is -1.
IloRound(0.6) is 1.
IloRound(-0.6) is -1.
IloRound(1e300) is 1e300.
IloRound(1.0000001e6) is 1e6.
IloRound(1.0000005e6) is 1.000001e6.
```

Global function IloInstantiate

```
public IloGoal IloInstantiate(const IloEnv env, const IloNumVar var)
public IloGoal IloInstantiate(const IloEnv env, const IloAnyVar var, const
IloAnyValueSelector select)
public IloGoal IloInstantiate(const IloEnv env, const IloAnyVar var)
public IloGoal IloInstantiate(const IloEnv env, const IloNumVar var, const
IloIntValueSelector select)
public IloGoal IloInstantiate(const IloEnv env, const IloIntSetVar var)
public IloGoal IloInstantiate(const IloEnv env, const IloAnySetVar var)
public IloGoal IloInstantiate(const IloEnv env, const IloIntSetVar var, const
IloIntSetValueSelector select)
public IloGoal IloInstantiate(const IloEnv env, const IloAnySetVar var, const
IloAnySetValueSelector select)
```

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal, using the optional numeric selector. If the variable has already been bound to a value, then this goal does nothing and succeeds. Otherwise, it sets a choice point and then alters the domain of the variable in recursive attempts to bind the variable. The way the goal alters the domain of the variable depends on the class of the variable and its optional selector. In other words, it tries values of the variable in an intelligent attempt to find a solution.

This function returns an instance of `IloGoal` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the goal that it returns as an instance of `IlcGoal` for use during a Solver search.

When the Variable Is an Integer or Enumerated Variable

When the variable is an integer variable or an enumerated variable, this function tries all values in the domain of the variable in the order indicated by the selector, removing those that fail at each trial, setting a choice point, and trying again recursively.

When the Variable Is a Set Variable

When the variable is a set variable, this function uses the selector to choose an element from the *possible* elements of the set to add to the *required* set of the variable. By default, the selector tries integers in ascending order. If failure occurs, the element is removed from the *possible* set, and another element is tried.

Differs from IloBestInstantiate

This goal differs from the goal returned by the function `IloBestInstantiate` because this goal will try all values in the domain of the variable according to the selector until it succeeds or until the domain is exhausted, whereas `IloBestInstantiate` tries only one.

> **Note**
>
> Though this function works on numerical variables of type `Float` and type `Int`, it is preferable to use the function `IloDichotomize` with floating-point variables.

<ilsolver/ilosolverint.>

<ilsolver/ilosolverset.>

See Also: IloBestInstantiate, IloDichotomize, IloGenerate, IloGoal, IlcInstantiate

Global function IloUnion

`public IloNumToAnySetStepFunction IloUnion(const IloNumToAnySetStepFunction f1, const IloNumToAnySetStepFunction f2)`

Definition file: ilconcert/ilosetfunc.h

Represents a function equal to the union of the functions.
This operator creates and returns a function equal to the union of the functions `f1` and `f2`. The argument functions `f1` and `f2` must be defined on the same interval. The resulting function is defined on the same interval as the arguments. See also: `IloNumToAnySetStepFunction`.

Global function IloUnion

```
public IloIntervalList IloUnion(const IloIntervalList intervals1, const
IloIntervalList intervals2)
```

Definition file: ilconcert/ilointervals.h

Creates and returns the union of two interval lists.
This operator creates and returns an interval list equal to the union of the interval lists `intervals1` and `intervals2`. The arguments `intervals1` and `intervals2` must be defined on the same interval. An instance of `IloException` is thrown if two intervals with different types overlap. The resulting interval list is defined on the same interval as the arguments. See also: `IloIntervalList`.

Global function IlcImpactVarEvaluator

```
public IloEvaluator< IlcIntVar > IlcImpactVarEvaluator(IloEnv env)
public IloEvaluator< IlcIntVar > IlcImpactVarEvaluator(IloSolver solver)
```

Definition file: ilsolver/custgoal.h
Include file: <ilsolver/ilosolver.h>

This function returns an instance of the evaluator `IloEvaluator<IlcIntVar>`. The evaluation returns the result of the function `solver.getImpact(x)`, where `x` is the evaluated variable.

Global function IloScanNHood

```
public IloGoal IloScanNHood(IloEnv env, IloNHood nhood, IloSolution solution,
IloInt minChunk=IloMinChunk, IloInt maxChunk=IloMaxChunk)
public IloGoal IloScanNHood(IloEnv env, IloNHood nhood, IloSolution solution,
IloSolutionDeltaCheck check, IloInt minChunk=IloMinChunk, IloInt
maxChunk=IloMaxChunk)
public IloGoal IloScanNHood(IloEnv env, IloNHood nhood, IloNeighborIdentifier nid,
IloSolution solution, IloInt minChunk=IloMinChunk, IloInt maxChunk=IloMaxChunk)
public IloGoal IloScanNHood(IloEnv env, IloNHood nhood, IloNeighborIdentifier nid,
IloSolution solution, IloSolutionDeltaCheck check, IloInt minChunk=IloMinChunk,
IloInt maxChunk=IloMaxChunk)
public IlcGoal IloScanNHood(IloSolver solver, IloNHood nhood, IloSolution solution,
IloInt minChunk=IloMinChunk, IloInt maxChunk=IloMaxChunk)
public IlcGoal IloScanNHood(IloSolver solver, IloNHood nhood, IloSolution solution,
IloSolutionDeltaCheck check, IloInt minChunk=IloMinChunk, IloInt
maxChunk=IloMaxChunk)
public IlcGoal IloScanNHood(IloSolver solver, IloNHood nhood, IlcNeighborIdentifier
nid, IloSolution solution, IloInt minChunk=IloMinChunk, IloInt
maxChunk=IloMaxChunk)
public IlcGoal IloScanNHood(IloSolver solver, IloNHood nhood, IlcNeighborIdentifier
nid, IloSolution solution, IloSolutionDeltaCheck check, IloInt
minChunk=IloMinChunk, IloInt maxChunk=IloMaxChunk)
```

Definition file: ilsolver/iimls.h
Include file: <ilsolver/iimls.h>

These goals scan the neighborhood `nhood`. Each leaf node of these goals corresponds to a legal neighbor of the current solution `solution`.

If specified, `nid` is used to hold the index and the delta of the neighbor (as with respect to `nh.define`) instantiated at the current leaf. This can be used to communicate with other goals such as `IloNotify` and `IloTest`.

If any applied delta is to be saved back to `solution`, this must be done explicitly via `solution.store(solver)`, or using `IloStoreSolution`.

The argument `check` is used to filter out any unwanted solution deltas received from the neighborhood before instantiation of constrained variables begins. Meta-heuristics automatically supply such a checking object via `IloMetaHeuristic::getDeltaCheck`. If check is not specified, no additional checking is performed.

See Also: IloApplyMetaHeuristic, IloMetaHeuristic, IloNHood, IloNotify, IloScanDeltas, IloSingleMove

Global function IlcChooseMinSizeAny

`public IlcInt IlcChooseMinSizeAny(const IlcAnyVarArray vars)`

Definition file: ilsolver/ilcany.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the smallest domain from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IlcAnyVarArray, IlcChooseAnyIndex

Global function IloChooseMinMinInt

`public IlcInt` **`IloChooseMinMinInt`**`(const IlcIntVarArray vars)`

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the smallest minimum from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseIntIndex, IloGenerate

Global function IloRemoveValue

`public IloGoal IloRemoveValue(const IloEnv env, const IloNumVar var, IloNum value)`

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal in the environment of the constrained numeric variable `var` and removes `value` as a possible value from the domain of `var`.

When it takes an instance of the class `IloEnv` as a parameter, it returns an instance of `IloGoal` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the goal that it returns as an instance of `IlcGoal` for use during a Solver search.

For more information, see `IloNullIntersect`.

This function works only on numerical variables of type `Int`.

See Also: IloGoal, IlcRemoveValue, IloNullIntersect

Global function IlcElementNEq

`public IlcConstraint` **`IlcElementNEq`**`(IlcInt val, IlcIntArray array, IlcIntVar index, const char * name=0)`

Definition file: ilsolver/ilcint.h
Include file: ilsolver/ilosolver.h

This function creates and returns a constraint that forces the the value `val` to not be equal to the element `index` of the array `array`.

Global function IlcElementNEq

```
public IlcConstraint IlcElementNEq(IlcIntVar var, IlcIntArray array, IlcIntVar
index, const char * name=0)
```

Definition file: ilsolver/ilcint.h
Include file: ilsolver/ilosolver.h

This function creates and returns a constraint that forces the the variable `var` to not be equal to the element `index` of the array `array`.

Global function IlcChooseMaxSizeInt

`public IlcInt IlcChooseMaxSizeInt(const IlcIntVarArray vars)`

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the greatest domain from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseIntIndex, IloGenerate

Global function IlcRadToDeg

`public IlcFloat IlcRadToDeg(IlcFloat angle)`

Definition file: ilsolver/nlinflt.h

This function converts an angle expressed in radians to an angle expressed in degrees.

If you want to manipulate constrained floating-point expressions in degrees, we strongly recommend that you call the trigonometric functions on variables expressed in radians and then convert the results to degrees (rather than declaring the constrained floating-point expressions in degrees and then converting them to radians to call the trigonometric functions).

The reason for that advice is that the method we recommend gives more accurate results in the context of the usual floating-point pitfalls. See the *IBM ILOG Solver User's Manual* for an explanation of those pitfalls.

Good practice:

```
IloSolver s;
IlcFloatVar x(s, 0,IlcPi);
IlcFloatExp y=IlcSin(x);
// ... constraints on x and y ...
```

Global function ilc_fail_stop_here

`public void` **`ilc_fail_stop_here`**`()`

Definition file: ilsolver/ilcerr.h
Include file: <ilsolver/ilosolver.h>

The function `ilc_fail_stop_here` is useful when you are running a Solver program from a debugger, such as `dbx`: you can stop the execution on failures. You can set a breakpoint for the function `ilc_fail_stop_here` to stop the program after each failure. You do that like this:

```
(dbx) stop in ilc_fail_stop_here
```

On some platforms, such as Windows NT, for example, you must prefix an underscore to the name of the function, like this: `_ilc_fail_stop_here`.

For more information, see the concept Failure.

See Also: IlcTrace, IlcTraceI

Global function IlcLog

```
public IlcFloatExp IlcLog(const IlcFloatExp x)
public IlcFloat IlcLog(IlcFloat x)
```

Definition file: ilsolver/nlinflt.h
Include file: <ilsolver/ilosolver.h>

This function creates a new constrained expression equal to the natural logarithm of the constrained expression `x`. The effects of this function are reversible. If its argument is an instance of `IlcFloat`, it simply returns the natural logarithm of its argument. With Solver, you should use this function instead of the standard C++ function `log`.

See Also: IlcExponent, IlcFloatExp, IlcMonotonicIncreasingFloatExp

Global function IloScalProd

```
public IloNum IloScalProd(const IloNumArray vals1, const IloNumArray vals2)
public IloNum IloScalProd(const IloIntArray vals1, const IloNumArray vals2)
public IloNum IloScalProd(const IloNumArray vals1, const IloIntArray vals2)
```

Definition file: ilconcert/iloexpression.h

Represents the scalar product.
This function returns a numeric value representing the scalar product of numeric values in the arrays `vals1` and `vals2`.

Global function IloScalProd

```
public IloNumExprArg IloScalProd(const IloNumArray values, const IloNumVarArray
vars)
public IloNumExprArg IloScalProd(const IloNumVarArray vars, const IloNumArray
values)
public IloNumExprArg IloScalProd(const IloNumArray values, const IloIntVarArray
vars)
public IloNumExprArg IloScalProd(const IloIntVarArray vars, const IloNumArray
values)
public IloNumExprArg IloScalProd(const IloIntArray values, const IloNumVarArray
vars)
public IloNumExprArg IloScalProd(const IloNumVarArray vars, const IloIntArray
values)
public IloNumExprArg IloScalProd(const IloNumExprArray leftExprs, const
IloNumExprArray rightExprs)
```

Definition file: ilconcert/iloexpression.h

Represents the scalar product.
This function returns an instance of `IloNumExprArg`, the internal building block of an expression, representing the scalar product of the variables in the arrays `vars` and `values` or the arrays `leftExprs` and `rightExprs`.

Global function IloScalProd

```
public IloIntExprArg IloScalProd(const IloIntArray values, const IloIntVarArray
vars)
public IloIntExprArg IloScalProd(const IloIntVarArray vars, const IloIntArray
values)
public IloIntExprArg IloScalProd(const IloIntExprArray leftExprs, const
IloIntExprArray rightExprs)
```

Definition file: ilconcert/iloexpression.h

Represents the scalar product.
This function returns an instance of `IloIntExprArg`, the internal building block of an integer expression, representing the scalar product of the variables in the arrays `vars` and `values` or the arrays `leftExprs` and `rightExprs`.

Global function IloScalProd

`public IloInt` **`IloScalProd`**`(const IloIntArray vals1, const IloIntArray vals2)`

Definition file: ilconcert/iloexpression.h

Represents the scalar product.
This function returns an integer value representing the scalar product of integer values in the arrays `vals1` and `vals2`.

Global function IloSolutionEvaluator

```
public IloEvaluator< IloSolution > IloSolutionEvaluator(IloEnv env, IloBoolVar var)
```

Definition file: ilsolver/iimpooleval.h
Include file: <ilsolver/iim.h>

Evaluates the value of a Boolean variable in a solution.
This function creates a solution evaluator on the environment `env` which evaluates the value of the Boolean variable `var` in the solution to be evaluated.

Global function IloSolutionEvaluator

```
public IloEvaluator< IloSolution > IloSolutionEvaluator(IloEnv env)
```

Definition file: ilsolver/iimpooleval.h
Include file: <ilsolver/iim.h>

Evaluates the objective value of a solution.
This function creates a solution evaluator on the environment `env` which evaluates the *objective value* of a solution.

Global function IloSolutionEvaluator

```
public IloEvaluator< IloSolution > IloSolutionEvaluator(IloEnv env, IloNumVar var)
```

Definition file: ilsolver/iimpooleval.h
Include file: <ilsolver/iim.h>

Evaluates the value of a floating point variable in a solution.
This function creates a solution evaluator on the environment `env` which evaluates the value of the floating-point variable `var` in the solution to be evaluated.

Global function IloSolutionEvaluator

```
public IloEvaluator< IloSolution > IloSolutionEvaluator(IloEnv env, IloIntVar var)
```

Definition file: ilsolver/iimpooleval.h
Include file: <ilsolver/iim.h>

Evaluates the value of an integer variable in a solution.
This function creates a solution evaluator on the environment `env` which evaluates the value of the integer variable `var` in the solution to be evaluated.

Global function operator>=

```
public IloConstraint operator>=(IloNumExprArg base, IloNumExprArg base2)
public IloRange operator>=(IloNumExprArg expr, IloNum val)
public IloRange operator>=(IloNum val, IloNumExprArg eb)
```

Definition file: ilconcert/ilolinear.h

overloaded C++ operator
This overloaded C++ operator constrains its first argument to be greater than or equal to its second argument. In order to be taken into account, this constraint must be added to a model and extracted for an algorithm.

Global function operator>=

```
public IlcBool operator>=(const IlcRevFloat & rev, IlcFloat value)
public IlcBool operator>=(const IlcRevInt & rev, IlcInt value)
public IlcBool operator>=(IlcInt value, const IlcRevInt & rev)
public IlcBool operator>=(const IlcRevInt & rev1, const IlcRevInt & rev2)
public IlcBool operator>=(IlcFloat value, const IlcRevFloat & rev)
public IlcBool operator>=(const IlcRevFloat & rev1, const IlcRevFloat & rev2)
public IlcBool operator>=(const IlcIntToIntStepFunction & f1, const
IlcIntToIntStepFunction & f2)
```

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>

This operator compares its arguments; if the first argument is greater than or equal to the second, then it returns `IlcTrue`; otherwise, it returns `IlcFalse`.

See Also: IlcRevFloat, IlcRevInt, IlcIntToIntStepFunction

Global function operator>=

```
public IloPredicate< IloObject > operator>=(IloEvaluator< IloObject > left,
IloEvaluator< IloObject > right)
public IloPredicate< IloObject > operator>=(IloEvaluator< IloObject > left, IloNum
threshold)
public IloPredicate< IloObject > operator>=(IloNum threshold, IloEvaluator<
IloObject > right)
```

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

Creates a greater-than-or-equal predicate from two evaluators.
These operators create a new `IloPredicate<IloObject>` instance by comparing the value returned by an evaluator with either that of another evaluator or a threshold value. The semantics of the new predicate are a greater-than-or-equal comparison. The first function creates a predicate which returns `IloTrue` if and only if the value returned by the left evaluator is greater than or equal to the value returned by the right evaluator. The second function creates a predicate which returns `IloTrue` if and only if the value returned by the left evaluator is greater than or equal to the threshold value. Finally, the third function creates a predicate that returns `IloTrue` if and only if the threshold value is greater than or equal to the value returned by the right evaluator.

For more information, see Selectors.

Global function operator>=

```
public IlcConstraint operator>=(const IlcIntExp exp1, const IlcIntExp exp2)
public IlcConstraint operator>=(const IlcIntExp exp1, IlcInt exp2)
public IlcConstraint operator>=(IlcInt exp1, const IlcIntExp exp2)
public IlcConstraint operator>=(const IlcFloatExp exp1, const IlcFloatExp exp2)
public IlcConstraint operator>=(const IlcFloatExp exp1, IlcFloat exp2)
public IlcConstraint operator>=(IlcFloat exp1, const IlcFloatExp exp2)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This operator creates and returns an inequality constraint between its arguments (that is, the first must be greater than or equal to the second).

When its arguments are constrained floating-point or integer expressions, then when it is posted, this constraint is associated with the `whenRange` propagation event.

When you create a constraint, it has no effect until you post it.

See Also: IlcConstraint, IlcFloatExp, IlcIntExp, IlcLeOffset, IlcNull, operator<=, operator!=, operator==

Global function IloMinimizeVar

```
public IloSearchSelector IloMinimizeVar(const IloEnv env, const IloNumVar var,
IloNum step=0)
```

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a search selector to act as a filter during the search for a solution in a Concert Technology model.

The search selector that this function creates and returns does several things:

- It stores the leaf of the search tree corresponding to the optimal value of the variable `var` and then reactivates this variable after the complete exploration of the search tree.
- It manages the upper bound on the objective variable. As soon as a solution of value `d` is found, the constraint `var <= d - step` is added to the model for the remainder of the search.
- Open nodes are evaluated. The *evaluation* of an open node is equal to the current minimum of the variable `var` when the node is created. When the search requires an open node, it checks whether the current upper bound on the objective is less than the evaluation of the node. If so, the node is safely discarded.

When this function takes an instance of the class `IloEnv` as a parameter, it returns an instance of `IloSearchSelector` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the search selector that it returns as an instance of `IlcSearchSelector` for use during a Solver search.

See Also: IloSearchSelector

Global function IlcElementEq

```
public IlcConstraint IlcElementEq(IlcIntVar var, IlcIntArray array, IlcIntVar
index, const char * name=0)
```

Definition file: ilsolver/ilcint.h
Include file: ilsolver/ilosolver.h

This function creates and returns a constraint that forces the the variable `var` to be equal to the element `index` of the array `array`.

Global function IlcElementEq

`public IlcConstraint IlcElementEq(IlcInt val, IlcIntArray array, IlcIntVar index, const char * name=0)`

Definition file: ilsolver/ilcint.h
Include file: ilsolver/ilosolver.h

This function creates and returns a constraint that forces the the value `val` to be equal to the element `index` of the array `array`.

Global function IloCompose

`public IloNHood **IloCompose**(IloEnv env, IloNHood nhood1, IloNHood nhood2, const char * name=0)`

Definition file: ilsolver/iimnhood.h
Include file: <ilsolver/ilosolver.h>

This function creates a *composed neighborhood.* The neighborhood formed is the product of `nhood1` and `nhood2`. Specifically, each neighbor specified by `nhood1` is composed with each neighbor specified by `nhood2`. The size of the resulting neighborhood is the product of the sizes of `nhood1` and `nhood2`. Each call to `define` creates a *composed delta* which is a *union* of the deltas of the corresponding neighbors of the composed neighborhoods. This union operates by placing in the composed delta all variables mentioned in either of two deltas from `nhood1` and `nhood2`. If the delta from `nhood1` and that from `nhood2` set different values for the same variable, the value in the delta from `nhood2` is used in the composed delta. The optional argument `name`, if provided, becomes the name of the newly created neighborhood.

Note

This neighborhood does *not* provide a chaining of moves (a move from `nhood1` followed by a move from `nhood2`). It provides simultaneous moves from `nhood1` and `nhood2`. In that sense, only moves from `nhood1` and `nhood2` that mention different parts of the solution will result in a sensible move; others which mention the same part of the solution will most likely create unwanted interactions and will typically be illegal.

In such a composed neighborhood, the member function `IloNHood::define` calls `IloNHood::define` on both `nhood1` and `nhood2`, returning the composed solution delta; the function `IloNHood::notify` notifies both `nhood1` and `nhood2`; the function `IloNHood::start` calls `IloNHood::start` for both `nhood1` and `nhood2`;and the function `IloNHood::getSize` returns the product of the sizes of `nhood1` and `nhood2`.

See Also: IloConcatenate, IloNHood, operator*

Global function IloSolveOnce

`public IloGoal IloSolveOnce(const IloEnv env, const IloGoal goal)`

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal in a Concert Technology model. This returned goal will solve `goal` (the one passed as an argument) in a nested search. This returned goal will search for exactly one solution of `goal` (the argument goal). Other solutions will not be tried. If the nested search fails, then the returned goal fails. If the nested search succeeds, the search will continue from the solution of the nested search.

This function returns an instance of `IloGoal` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the goal that this function returns as an instance of `IlcGoal` for use during a Solver search.

See Also: IlcGoal, IloGoal, IloSolver

Global function IlcOnce

`public IlcGoal IlcOnce(IlcGoal goal)`

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal. This returned goal will solve `goal` (the one passed as an argument) in a nested search. This returned goal will search for exactly one solution of `goal` (the argument goal). Other solutions will not be tried. If the nested search fails, then the returned goal fails. If the nested search succeeds, the search will continue from the solution of the nested search.

This function returns an instance of `IlcGoal` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`.

See Also: IlcGoal

Global function IloTestDelta

`public IlcGoal` **`IloTestDelta`**`(IloSolver solver, IloSolution solution, IloSolution delta)`
`public IloGoal` **`IloTestDelta`**`(IloEnv env, IloSolution solution, IloSolution delta)`

Definition file: ilsolver/iimls.h
Include file: <ilsolver/iimls.h>

These functions take a solution `solution` and a delta `delta`, and return a goal that:

1. Restores `delta`.
2. Restores the part of `solution` that does not appear in `delta`.

The goal can fail at 1) or 2) if any constraints are violated, that is, the goal fails if the proposed change to the solution is illegal. If changes made are to be recorded back into `solution`, then this process must be performed afterwards, for example, using `solution.store(solver)`. Alternatively, you can use `IloCommitDelta`.

Implementation

The execute method for a goal that behaves in this way can be implemented as follows:

```
IlcGoal IloTestDelta::execute() {
  delta.restore(solver);
  solution.restore(solver, delta);
  return 0;
}
```

For more information, see `IloSolution` in the *Concert Technology Reference Manual*.

See Also: IloCommitDelta, IloStoreSolution

Global function IloEqPartition

```
public IloConstraint IloEqPartition(const IloEnv env, const IloAnySetVar var, const
IloAnySetVarArray vars)
public IloConstraint IloEqPartition(const IloEnv, const IloIntSetVar var, const
IloIntSetVarArray vars)
```

Definition file: ilconcert/iloanyset.h

For IBM® ILOG® Solver: a constraint forcing the value of a variable to be required by one set variable in an array.
These functions create and return a constraint. When that constraint is posted, it insures that the value of the variable `var` will be required by exactly one set variable of the array `vars`. In this context, a constraint will be posted after it has been added to a model and extracted by a solver (for example, an instance of `IloSolver` documented in the *IBM ILOG Solver Reference Manual*).

Global function IloSetMin

`public IloGoal IloSetMin(const IloEnv env, const IloNumVar var, IloNum value)`

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal for the constrained numeric variable `var` with `value` as its minimum.

When it takes an instance of the class `IloEnv` as a parameter, it returns an instance of `IloGoal` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the goal that it returns as an instance of `IlcGoal` for use during a Solver search.

For more information, see `IloNullIntersect`.

This function works on numerical variables of type `Float` and type `Int`.

See Also: IloGoal, IlcSetMin, IloNullIntersect

Global function IloInitMT

```
public void IloInitMT()
public void IloInitMT(IloBaseEnvMutex *)
```

Definition file: ilconcert/iloenv.h

Initializes multithreading.
This function initializes multithreading in a Concert Technology application.

Global function IlcLeOffset

`public IlcConstraint IlcLeOffset(const IlcIntExp x, const IlcIntExp y, IlcInt c)`

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a constraint equivalent to `x <= y + c`. That constraint is violated when the minimum of `x` is greater than the maximum of `y + c`.

When it is posted, this constraint reacts to the `whenRange` propagation event.

See Also: IlcConstraint, IlcIntExp, operator==, operator<=, operator>=, operator<, operator>

Global function IlcInverse

`public IlcConstraint IlcInverse(IlcIntSetVarArray f, IlcIntVarArray invf)`

Definition file: ilsolver/ilcset.h

An instance of `IlcInverse` represents an inverse constraint between an array on set variables and an array of variables. Informally, we say that an inverse constraint works on two arrays, say, `f` and `invf`, so that an element of `f` collects the indexes of the element of `invf` that points to the first element.

In formal terms, if the length of the array `f` is `n`, and the length of the array `invf` is `m`, then the inverse constraint makes sure that:

- for all `i` in the interval `[0, n-1]` and
- for all `j` in the interval `[0, m-1]`, then
- `j` is in `f[i]` is equivalent to `invf[j] == i`.

See Also: IlcConstraint

Global function IlcInverse

`public IlcConstraint IlcInverse(IlcIntVarArray f, IlcIntVarArray invf)`

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

If the length of the array `f` is `n`, and the length of the array `invf` is `m`, then this function returns a constraint that insures that:

- for all `i` in the interval `[0, n-1]`, if `f[i]` is in `[0, m-1]` then `invf[f[i]] == i`;
- for all `j` in the interval `[0, m-1]`, if `invf[j]` is in `[0, n-1]` then `f[invf[j]] == j`.

See Also: IlcConstraint, IlcIntVarArray, IlcPathLength

Global function IlcMin

```
public IlcIntExp IlcMin(const IlcIntSetVar aSet, const IlcBool
makeEmptySetPropagation=IlcTrue)
```

Definition file: ilsolver/setcst.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a new constrained expression equal to the least of the integer elements that are assigned to the constrained set variable `setVar`.

$$y = \underset{x \in \text{setVar}}{\text{MIN}} \; x$$

The minimum value of the expression is the least possible integer element of the variable `setVar`.

The maximum value of the expression is the least required integer element of the variable `setVar`, or, if there is none, then the greatest possible element.

Because the minimum value of an empty set has no meaning, the bounds of this expression are computed only when the set variable is surely not empty, that is, when its cardinality is greater than 0 (zero). The initial bounds of the expression are the minimum and the maximum possible elements.

See Also: IlcIntExp, IlcIntSetVar

Global function IlcMin

```
public IlcIntExp IlcMin(const IlcIntSetVar aSet, const IlcIntToIntFunction F, const
IlcBool makeEmptySetPropagation=IlcTrue)
public IlcIntExp IlcMin(const IlcIntSetVar aSet, const IlcIntToIntExpFunction F,
const IlcBool makeEmptySetPropagation=IlcTrue)
public IlcIntExp IlcMin(const IlcAnySetVar aSet, const IlcAnyToIntFunction F, const
IlcBool makeEmptySetPropagation=IlcTrue)
public IlcIntExp IlcMin(const IlcAnySetVar aSet, const IlcAnyToIntExpFunction F,
const IlcBool makeEmptySetPropagation=IlcTrue)
```

Definition file: ilsolver/setcst.h
Include file: <ilsolver/ilosolver.h>

These functions create and return a new constrained expression equal to the least of the values returned by the function `F` applied to the elements assigned to the constrained set variable `setVar`.

$$y = \min_{x \in \mathrm{setVar}} F(x)$$

The value returned by the function `F` can be an integer value (`IlcInt`) or an integer constrained expression (`IlcIntExp`).

The minimum value of the expression is the least value returned by `F` when applied to the possible elements of the variable `setVar`. If `F` returns an integer expression, it corresponds to the least lower bound of `F(x)`.

The maximum value of the expression is the least value returned by `F` when applied to the required elements of the variable `setVar`. If there is no required element, it corresponds to the greatest value returned by `F` when applied to the possible elements of `setVar`. When `F` returns an integer expression, the maximum value of the expression is computed with the upper bounds of `F(x)`.

Because the minimum value of an empty set has no meaning, the bounds of this expression are computed only when the set variable is surely not empty, that is, when its cardinality is greater than 0 (zero). The initial bounds of the expression are the minimum value and the maximum value returned by `F` when applied to the possible elements of `setVar`.

See Also: IlcAnySetVar, IlcIntExp, IlcIntSetVar

Global function IlcMin

```
public IlcIntExp IlcMin(IlcInt exp1, const IlcIntExp exp2)
public IlcFloat IlcMin(const IlcFloat exp1, const IlcFloat exp2)
public IlcIntExp IlcMin(const IlcIntExp exp1, IlcInt exp2)
public IlcIntExp IlcMin(const IlcIntExp exp1, const IlcIntExp exp2)
public IlcIntExp IlcMin(const IlcIntVarArray array)
public IlcInt IlcMin(const IlcIntArray array)
public IlcInt IlcMin(const IlcInt exp1, const IlcInt exp2)
public IlcFloatExp IlcMin(const IlcFloatExp exp1, IlcFloat exp2)
public IlcFloatExp IlcMin(IlcFloat exp1, const IlcFloatExp exp2)
public IlcFloatExp IlcMin(const IlcFloatExp exp1, const IlcFloatExp exp2)
public IlcFloatExp IlcMin(const IlcFloatVarArray array)
public IlcFloat IlcMin(const IlcFloatArray array)
public IlcIntToIntStepFunction IlcMin(const IlcIntToIntStepFunction & f1, const
IlcIntToIntStepFunction & f2)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function returns the minimum of its argument or arguments.

When its argument is an array of constrained expressions, this function creates a new constrained expression equal to the minimum of its argument.

When at least one of its arguments is a constrained expression, this function creates a new constrained expression equal to the minimum of its arguments.

When both its arguments are numeric (that is, values of type `IloInt` or `IloNum`), it simply creates and returns a value of that type.

See Also: IlcFloatExp, IlcFloatVarArray, IlcIntExp, IlcIntToIntStepFunction, IlcIntVarArray, IlcMax

Global function IlcMin

```
public IlcFloatExp IlcMin(const IlcIntSetVar aSet, const IlcIntToFloatExpFunction
f, const IlcBool makeEmptySetPropagation=IlcTrue)
```

Definition file: ilsolver/setcst.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a floating-point expression constrained to be the minimum returned by the function `f` over its domain, `aSet`.

See Also: IlcIntSetVar

Global function IloDiv

```
public IloIntExprArg IloDiv(const IloIntExprArg x, const IloIntExprArg y)
public IloIntExprArg IloDiv(const IloIntExprArg x, IloInt y)
public IloIntExprArg IloDiv(IloInt x, const IloIntExprArg y)
```

Definition file: ilconcert/iloexpression.h

Integer division function.
This function is available for integer division. For numeric division, use operator/.

Global function IloUpdateBestSolution

`public IlcGoal` **`IloUpdateBestSolution`**`(IloSolver solver, IloSolution best, IloSolution soln)`
`public IloGoal` **`IloUpdateBestSolution`**`(IloEnv env, IloSolution best, IloSolution current)`

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function returns a goal that copies the values of variables in `current` to those in `best` if the value of the objective of `current` is better than that of `best`. What is considered better depends on the sense of the objective added to `solution`. This goal always succeeds.

This function can be implemented as follows:

```
IlcGoal IlcUpdateBestSolutionI::execute() {
  if (current.isBetterThan(best)) best.copy(current);
  return 0;
}
```

For more information, see `IloSolution`.

See Also: IloRestoreSolution, IloSolution, IloStoreBestSolution, IloStoreSolution

Global function IlcChooseMaxMinFloat

```
public IlcInt IlcChooseMaxMinFloat(const IlcFloatVarArray vars)
```

Definition file: ilsolver/linfloat.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the greatest minimum from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseFirstUnboundFloat, IlcFloatVarArray, IloGenerate

Global function IlcEqBoolAbstraction

```
public IlcConstraint IlcEqBoolAbstraction(IlcBoolVarArray ys, IlcAnyVarArray xs,
IlcAnyArray vals)
public IlcConstraint IlcEqBoolAbstraction(IlcBoolVarArray ys, IlcIntVarArray xs,
IlcIntArray vals)
```

Definition file: ilsolver/ilcany.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a constraint that `ys[i]` is the Boolean abstraction of `xs[i]` with respect to `vals` for use in a Solver search.

The argument `xs` should be an array of constrained variables. The argument `vals` is an array of integers or pointers.

For each `xs[i]`, Solver creates the *Boolean abstraction* of `xs[i]` with respect to `vals`. In other words, for every variable in `xs`, `xs[i]`, Solver creates a Boolean variable `ys[i]` such that `ys[i]=0` if and only if `xs[i]` cannot be bound with a value of `vals`, and `ys[i]=1` if and only if `xs[i]` will necessarily be bound with a value of `vals`. Then Solver insures that this property holds after the definition of the Boolean abstraction.

For a function that returns the array `ys`, rather than a constraint, see `IlcBoolAbstraction`.

For a constraint suitable for use in a *model*, see `IloBoolAbstraction`.

See Also: IlcAbstraction, IlcBoolAbstraction, IloBoolAbstraction

Global function IlcCard

`public IlcIntExp IlcCard(IlcIndex & i, IlcConstraint ct)`

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

You use the function `IlcCard` to control the size of a *set of constraints* during a Solver search. For similar functionality in a model, consider `IloCard` documented in the IBM ILOG Concert API.

This function creates and returns a constrained integer expression. At all times, that expression is greater than or equal to the number of integer values *j* such that the constraint `ct` is true for *j*. That expression is also less than or equal to the number of integer values *j* such that the constraint `ct` is not false for *j*.

The generic constraint `ct` must have been created with generic variables stemming from the index `i` (an argument passed to `IlcCard`); otherwise, Solver will throw an exception (an instance of `IloSolver::SolverErrorException`).

All generic variables of the constraint `ct` must represent arrays of constrained expressions of the same size; otherwise, Solver will throw an exception (an instance of `IloSolver::SolverErrorException`).

To count the number of occurrences of several values, use the function `IlcDistribute`.

Generic Constraints

A *generic constraint* is a constraint shared by an array of variables. For example, `IlcAllDiff` is a generic constraint that insures that all the elements of a given array are different from one another. Solver provides generic constraints to save memory since, if you use them, you can avoid allocating one object per variable.

You create a generic constraint simply by stating the constraint over *generic variables*. Each generic variable stands for all the elements of an array of constrained variables.

In that sense, generic variables are only syntactic objects provided by Solver to support generic constraints, and they can be used only for creating generic constraints. To create a generic variable, you use the operator `[]`. The argument passed to that operator is known as the *index* for that generic variable; we say that the generic variable *stems from* that index.

Implementation

This function can be implemented like this:

```
IlcIntExp IlcCard(IlcIndex& i, IlcConstraint ct) {
    return IlcCard(IlcSetOf(i, ct));
}
```

Example

Here's how to count the number of expressions in two arrays of constrained integer variables `x` and `y` such that `x[I] == y[I] + 2`:

```
IlcIndex I(s);
IlcIntVar number = IlcCard(I, x[I] == y[I] + 2);
```

A very common use of generic constraints is to put limits on the number of times that a value can appear in a given array of variables. For example, we could use the class `IlcIndex` and the function `IlcCard` to define a function `IlcCount` like this:

```
IlcConstraint IlcCount(IloSolver s,
                       IlcIntVar card,
                       IlcInt val,
                       IlcIntVarArray vars){
    IlcIndex i(s);
    return card == IlcCard(i, vars[i] == val);
}
```

If that constraint is posted, then it constrains `card` to be equal to the number of occurrences of `val` in the arrays `vars`. At any given moment, the minimum of `card` is at least equal to the number of variables contained in `vars` bound to the value `val`; and the maximum of `card` is at most equal to the number of variables contained in `vars` that contain `val` in their domain.

See Also: IlcCard, IloCard, IlcConstraint, IlcDistribute, IlcIndex, IlcIntExp, IlcSetOf

Global function IlcCard

```
public IlcIntVar IlcCard(IlcAnySetVar var)
public IlcIntExp IlcCard(IlcIntSetVar set)
```

Definition file: ilsolver/ilcset.h
Include file: <ilsolver/ilosolver.h>

A constrained set variable contains a constrained integer variable (called the cardinality variable) that represents the cardinality of the value of the constrained set variable. You can constrain that cardinality and thus control the size of a *constrained set variable*.

This function returns a constrained integer variable which is constrained to be equal to the cardinality of the domain of the invoking constrained set variable. Every call to `IlcCard` returns the *same* constrained integer variable for use during a Solver search. For similar functionality in a model, consider `IloCard` documented in the Concert API.

The size of the required set of the constrained set variable is less than or equal to the minimum of the domain of the constrained integer variable. The maximum of the domain of the constrained integer variable is less than or equal to the size of the possible set. More formally,

```
cardinal(required) <= min(var) <= max(var) <= cardinal(possible)
```

Examples:

Here's what we write in order to constrain the value of a given constrained set variable of pointers, `setVar,` to contain at least four elements:

```
s.add(IlcCard(setVar) >= 4); // setVar defined before
```

Here's what we write in order to constrain the value of a given constrained set variable of integers, `setVar`, to contain at least four elements:

```
s.add(IlcCard(setVar) >= 4); // setVar defined before
```

See Also: IlcAnySetVar, IlcCard, IlcIntSetVar, IlcIntVar, IlcSequence, IloCard

Global function IlcChooseMinSizeIntSet

`public IlcInt IlcChooseMinSizeIntSet(const IlcIntSetVarArray)`

Definition file: ilsolver/intset.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the smallest domain from among the constrained variables in the array of constrained variables `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseIntSetIndex, IloGenerate

Global function IlcSetValue

```
public IlcGoal IlcSetValue(const IlcIntVar var, const IlcInt val)
public IlcGoal IlcSetValue(const IlcFloatVar var, const IlcFloat val)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function returns a goal which sets the value of `var` to be `val`.

See Also: IlcRemoveValue

Global function IlcChooseMinMaxInt

`public IlcInt **IlcChooseMinMaxInt**(const IlcIntVarArray vars)`

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the smallest maximum from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseIntIndex, IloGenerate

Global function operator<

```
public IloPredicate< IloObject > operator<(IloEvaluator< IloObject > left,
IloEvaluator< IloObject > right)
public IloPredicate< IloObject > operator<(IloEvaluator< IloObject > left, IloNum
threshold)
public IloPredicate< IloObject > operator<(IloNum threshold, IloEvaluator<
IloObject > right)
```

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

Creates a less-than predicate from two evaluators.
These operators create a new IloPredicate<IloObject> instance by comparing the value returned by an evaluator with either that of another evaluator or a threshold value. The semantics of the new predicate is a less-than comparison. The first function creates a predicate which returns IloTrue if and only if the value returned by the left evaluator is less than the value returned by the right evaluator. The second function creates a predicate which returns IloTrue if and only if the value returned by the left evaluator is less than the threshold value. The third function creates a predicate that returns IloTrue if and only if the threshold value is less than the value returned by the right evaluator.

For more information, see Selectors.

Global function operator<

```
public IlcBool operator<(const IlcRevFloat & rev, IlcFloat value)
public IlcBool operator<(const IlcRevInt & rev, IlcInt value)
public IlcBool operator<(IlcInt value, const IlcRevInt & rev)
public IlcBool operator<(const IlcRevInt & rev1, const IlcRevInt & rev2)
public IlcBool operator<(IlcFloat value, const IlcRevFloat & rev)
public IlcBool operator<(const IlcRevFloat & rev1, const IlcRevFloat & rev2)
```

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>

This operator compares its arguments; if the first argument is strictly less than the second, then it returns `IlcTrue`; otherwise, it returns `IlcFalse`.

See Also: IlcRevFloat, IlcRevInt

Global function operator<

```
public IloConstraint operator<(IloNumExprArg base, IloNumExprArg base2)
public IloConstraint operator<(IloNumExprArg base, IloNum val)
public IloConstraint operator<(IloNum val, const IloNumExprArg expr)
public IloConstraint operator<(IloIntExprArg base, IloIntExprArg base2)
public IloConstraint operator<(IloIntExprArg base, IloInt val)
```

Definition file: ilconcert/ilolinear.h

overloaded C++ operator
This overloaded C++ operator constrains its first argument to be strictly less than its second argument. In order to be taken into account, this constraint must be added to a model and extracted for an algorithm.

Global function operator<

```
public IlcConstraint operator<(const IlcIntExp exp1, const IlcIntExp exp2)
public IlcConstraint operator<(const IlcIntExp exp1, IlcInt exp2)
public IlcConstraint operator<(IlcInt exp1, const IlcIntExp exp2)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This operator creates and returns an inequality constraint between its arguments (that is, the first must be strictly less than the second).

When its arguments are integer expressions, then when you post it, this constraint is associated with the `whenRange` propagation event.

When you create a constraint, it has no effect until you post it.

See Also: IlcConstraint, IlcIntExp, IlcLeOffset, operator>, operator<=, operator>=, operator!=, operator==

Global function IloLexicographic

```
public IloConstraint IloLexicographic(IloEnv env, IloIntExprArray x,
IloIntExprArray y, const char *=0)
```

Definition file: ilconcert/ilomodel.h

Returns a constraint which maintains two arrays to be lexicographically ordered.
The `IloLexicographic` function returns a constraint which maintains two arrays to be lexicographically ordered.

More specifically, `IloLexicographic(x, y)` maintains that `x` is less than or equal to `y` in the lexicographical sense of the term. This means that either both arrays are equal or that there exists `i < size(x)` such that for all `j < i`, `x[j] = y[j]` and `x[i] < y[i]`.

Note that the size of the two arrays must be the same.

Global function IlcDichotomize

`public IlcGoal **IlcDichotomize**(const IlcIntVar var, IlcBool increaseMin=IlcTrue)`

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal, a primitive in the algorithms that search for solutions. It is used to assign a value to a constrained variable. It handles constrained integer variables in the way that the function `IlcInstantiate` handles constrained floating-point variables by recursively searching half of the domain of its argument.

If its argument `var` has already been bound, then `IlcDichotomize` does nothing and succeeds. Otherwise, `IlcDichotomize` sets a choice point, then replaces the domain of `var` by one of the halves, and calls itself recursively. If failure occurs then, the domain is replaced by the other half, and `IlcDichotomize` is called recursively.

The optional argument `increaseMin` must be either `IlcTrue` or `IlcFalse`. If it is `IlcTrue`, then the upper half of the domain is tried first; otherwise, the lower half is tried first.

See Also: IlcBestInstantiate, IlcGoal, IlcInstantiate, IlcIntVar, IlcSplit

Global function IloCard

`public IloIntVar` **`IloCard`**`(IloAnySetVar vars)`

Definition file: ilconcert/iloanyset.h

For constraint programming: creates and returns a constrained numeric variable that represents the number of elements in `vars`.
This function creates and returns a constrained numeric variable that represents the number of elements in vars. In other words, it constrains the cardinality of a set variable.

For example, to constrain mySet to contain four or more elements, you can use `IloCard` in the following way:

```
model.add (IloCard (mySet) >= 4);
```

Global function IlcGenerateBounds

```
public IlcGoal IlcGenerateBounds(IlcFloatVar var, IlcFloat prec=.1)
public IlcGoal IlcGenerateBounds(IlcFloatVarArray array, IlcFloat prec=0.1)
```

Definition file: ilsolver/nlinflt.h
Include file: <ilsolver/ilosolver.h>

The function `IlcGenerateBounds` creates a goal that will try to reduce the bounds of variables.

For more information, see the concept Propagation.

See Also: IlcFloatVar, IlcFloatVarArray, IlcSolveBounds

Global function IlcGeLex

`public IlcConstraint IlcGeLex(IlcIntVarArray x, IlcIntVarArray y)`

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

The `IlcGeLex` function returns a constraint which maintains two arrays to be lexicographically ordered.

More specifically, `IlcGeLex(x, y)` maintains that `x` is greater than or equal to `y` in the lexicographical sense of the term. This mean that either both arrays are equal or that there exists `i < size(x)` such that for all `j < i`, `x[j] = y[j]` and `x[i] > y[i]`.

Note that the size of the two arrays must be the same.

See Also: IloLeLex, IlcLeLex, IloGeLex

Global function IlcSelectSearch

```
public IlcGoal IlcSelectSearch(IlcGoal goal, IlcSearchSelector selector)
```

Definition file: ilsolver/search.h
Include file: <ilsolver/ilosolver.h>

This goal applies the selector `selector` to the search tree defined by the goal `goal`. As the goal manager explores the search tree, it gives successful leaves to the selector. When the tree has been fully explored, the selector is called, and it re-activates the selected nodes.

See Also: IlcSearchSelector

Global function IloMember

```
public IloConstraint IloMember(const IloEnv env, const IloAnyVar element, const
IloAnySetVar setVar)
```

Definition file: ilconcert/iloanyset.h

show solver

Global function IloMember

```
public IloConstraint IloMember(const IloEnv, const IloNumExprArg expr, const
IloNumArray elements)
public IloConstraint IloMember(const IloEnv, const IloIntExprArg expr, const
IloIntArray elements)
public IloConstraint IloMember(const IloEnv, const IloIntVar var1, const
IloIntSetVar var2)
public IloConstraint IloMember(const IloEnv, IloInt var1, const IloIntSetVar var2)
```

Definition file: ilconcert/ilomodel.h

For constraint programming: creates and returns a constraint forcing `element` to be a member of `setVar`. This function creates and returns a constraint (an instance of `IloConstraint`) for use in a model. The constraint forces `expr` to be a member of `elements`.

In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

Global function IloMember

```
public IloConstraint IloMember(const IloEnv env, IloAny element, const IloAnySetVar
setVar)
```

Definition file: ilconcert/iloanyset.h

show solver

Global function IlcChooseMaxMaxFloat

```
public IlcInt IlcChooseMaxMaxFloat(const IlcFloatVarArray vars)
```

Definition file: ilsolver/linfloat.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the greatest maximum from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseFloatIndex, IlcFloatVarArray, IloGenerate

Global function IloArcCos

```
public IloNumExprArg IloArcCos(const IloNumExprArg arg)
public IloNum IloCos(IloNum val)
public IloNum IloSin(IloNum val)
public IloNum IloTan(IloNum val)
public IloNum IloArcCos(IloNum val)
public IloNum IloArcSin(IloNum val)
public IloNum IloArcTan(IloNum val)
public IloNumExprArg IloSin(const IloNumExprArg arg)
public IloNumExprArg IloCos(const IloNumExprArg arg)
public IloNumExprArg IloTan(const IloNumExprArg arg)
public IloNumExprArg IloArcSin(const IloNumExprArg arg)
public IloNumExprArg IloArcTan(const IloNumExprArg arg)
```

Definition file: ilconcert/iloexpression.h

Trigonometric functions.
Concert Technology offers predefined functions that return an expression from a trigonometric function on an expression. These predefined functions also return a numeric value from a trigonometric function on a numeric value as well.

Programming Hint

If you want to manipulate constrained floating-point expressions in degrees, we strongly recommend that you call the trigonometric functions on variables expressed in radians and then convert the results to degrees (rather than declaring the constrained floating-point expressions in degrees and then converting them to radians to call the trigonometric functions).

The reason for that advice is that the method we recommend gives more accurate results in the context of the usual floating-point pitfalls.

Global function IlcSetMin

```
public IlcGoal IlcSetMin(const IlcIntVar var, const IlcInt val)
public IlcGoal IlcSetMin(const IlcFloatVar, const IlcFloat)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function returns a goal which sets the minimum of `var` to be `val`.

See Also: IlcSetMax

Global function IloChooseMinRegretMax

`public IlcInt IloChooseMinRegretMax(const IlcIntVarArray vars)`

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the minimal difference between the maximal possible value and the next maximal possible value from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is based on the principle of *regret*. Regret is the difference between what would have been the best possible decision in a scenario and what was the actual decision.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseIntIndex, IloGenerate

Global function IloConcatenate

```
public IloNHood IloConcatenate(IloEnv env, IloNHoodArray nhoods, const char *
name=0)
```

Definition file: ilsolver/iimnhood.h
Include file: <ilsolver/iimls.h>

This function creates a *concatenated neighborhood*. The neighborhood formed is the concatenation of all the neighborhoods in `nhoods`. The optional argument `name`, if provided, becomes the name of the newly created neighborhood.

In a concatenated neighborhood, the indices of the newly created neighborhood map directly to indices in the sub-neighborhoods, according to the order in which the sub-neighborhoods are specified. For example, suppose we concatenate neighborhoods *n1*, *n2*, and *n3*, with sizes *s1*, *s2*, and *s3* respectively. The newly created neighborhood *n* will have size *s1*+*s2*+*s3*, with indices 0 to *s1*-1 corresponding to neighborhood *n1*, *s1* to *s1*+*s2*-1 corresponding to neighborhood *n2*, and the remainder of the indices corresponding to neighborhood *n3*.

In such a neighborhood, the member function `IloNHood::define` calls `define` on the appropriate sub-neighborhood with the index properly adjusted as described above; the function `IloNHood::notify` performs notification to the appropriate sub-neighborhood with the index likewise adjusted; the function `IloNHood::start` calls `start` for each sub-neighborhood; and the function `IloNHood::getSize` adds together all sizes returned by the sub-neighborhoods.

In concatenated neighborhoods, only the basic neighborhood that defined the delta for the move to be taken can have `IloNHood::notify` called. For all other basic neighborhoods, `IloNHood#notifyOther` is called with the delta for the move to be taken.

See Also: IloCompose, IloNHood, operator+

Global function IloApplyMetaHeuristic

`public IlcGoal` **`IloApplyMetaHeuristic`**`(IloSolver solver, IloSolution solution, IloMetaHeuristic mh, IlcGoal nhoodScanGoal)`
`public IloGoal` **`IloApplyMetaHeuristic`**`(IloEnv env, IloSolution solution, IloMetaHeuristic mh, IloGoal nhoodScanGoal)`

Definition file: ilsolver/iimls.h
Include file: <ilsolver/iimls.h>

This function applies a metaheuristic filter `mh` to the nodes of a goal `nhoodScanGoal` which is used for exploring possible moves for a local search procedure (for example, `IloScanNHood` or `IloScanDeltas`). The function returns a new goal with the metaheuristic filter applied.

Implementation

`IloApplyMetaHeuristic` can be implemented like this:

```
IlcGoal IloApplyMetaHeuristic(IloSolver solver,
                              IloSolution solution,
                              IloMetaHeuristic mh,
                              IlcGoal nhoodScanGoal) {
  return IlcAnd(IloStart(solver, mh, solution),
                nhoodScanGoal,
                IloTest(solver, mh));
}
```

or this:

```
IloGoal IloApplyMetaHeuristic(IloEnv env,
                              IloSolution solution,
                              IloMetaHeuristic mh,
                              IloGoal nhoodScanGoal) {
  return IloStart(env, mh, solution) && nhoodScanGoal
                                     && IloTest(env, mh);
}
```

See Also: IloMetaHeuristic, IloNotify, IloScanDeltas, IloScanNHood, IloSingleMove, IloStart, IloTest

Global function IloAbs

```
public IloNumExprArg IloAbs(const IloNumExprArg arg)
public IloNum IloAbs(IloNum val)
public IloNum IloPower(IloNum val1, IloNum val2)
public IloIntExprArg IloAbs(const IloIntExprArg arg)
```

Definition file: ilconcert/iloexpression.h

Returns the absolute value of its argument.
Concert Technology offers predefined functions that return an expression from an algebraic function on expressions. These predefined functions also return a numeric value from an algebraic function on numeric values as well.

`IloAbs` returns the absolute value of its argument.

What Is Extracted

`IloAbs` is extracted by an instance of `IloCplex` and linearized automatically.

`IloAbs` is extracted by an instance of `IloCP` or `IloSolver` as an instance of `IlcAbs`.

Global function IloSingleMove

```
public IlcGoal IloSingleMove(IloSolver solver, IloSolution s, IloNHood nh,
IloMetaHeuristic mh, IlcSearchSelector sel, IlcGoal subgoal, IlcNeighborIdentifier
nid, IloInt minChunk=IloMinChunk, IloInt maxChunk=IloMaxChunk)
public IloGoal IloSingleMove(IloEnv env, IloSolution soln, IloNHood nhood,
IloMetaHeuristic mh, IloSearchSelector sel, IloGoal subGoal, IloNeighborIdentifier
nid, IloInt minChunk=IloMinChunk, IloInt maxChunk=IloMaxChunk)
public IloGoal IloSingleMove(IloEnv env, IloSolution soln, IloNHood nhood,
IloMetaHeuristic mh, IloSearchSelector sel, IloGoal subGoal, IloInt
minChunk=IloMinChunk, IloInt maxChunk=IloMaxChunk)
public IloGoal IloSingleMove(IloEnv env, IloSolution solution, IloNHood nh)
public IloGoal IloSingleMove(IloEnv env, IloSolution solution, IloNHood nh,
IloMetaHeuristic mh, IloSearchSelector sel)
public IloGoal IloSingleMove(IloEnv env, IloSolution solution, IloNHood nh,
IloMetaHeuristic mh)
public IloGoal IloSingleMove(IloEnv env, IloSolution solution, IloNHood nh,
IloSearchSelector sel)
public IloGoal IloSingleMove(IloEnv env, IloSolution solution, IloNHood nh, IloGoal
subGoal)
public IloGoal IloSingleMove(IloEnv env, IloSolution solution, IloNHood nh,
IloMetaHeuristic mh, IloGoal subGoal)
public IloGoal IloSingleMove(IloEnv env, IloSolution solution, IloNHood nh,
IloSearchSelector sel, IloGoal subGoal)
public IlcGoal IloSingleMove(IloSolver solver, IloSolution solution, IloNHood
nhood, IloMetaHeuristic mh, IlcSearchSelector sel, IlcGoal subgoal, IloInt
minChunk=IloMinChunk, IloInt maxChunk=IloMaxChunk)
```

Definition file: ilsolver/iimls.h
Include file: <ilsolver/iimls.h>

This function returns a goal that makes a single local move as defined by a neighborhood, a metaheuristic, and a move selection method. The goal scans the neighborhood `nhood` using `solution` as the current solution. The metaheuristic `mh` is used to filter moves, and the search selector `sel` to choose a single move from those that become available. Additionally, a goal `subgoal` can be executed after the deltas from the neighborhood are applied to the current solution. A neighbor identifier `nid` can be specified which can, on successful completion of a move, be used to provide information about the neighbor.

If no metaheuristic is specified, no metaheuristic filtering is performed. If no selector is specified, `IloFirstSolution` is assumed. If no subgoal is specified, none is executed.

If a successful move can be found, the goal succeeds. The constrained variables are in the state corresponding to the application of the move. This new state is saved to `solution` before the goal succeeds. If no successful move can be found, the goal fails, and `solution` is left unchanged.

When `IloSingleMove` is used, the `IloMetaHeuristic::test` and `IloMetaHeuristic::notify` methods of the supplied metaheuristic are supplied with the relevant solution delta, and the neighborhood is notified with the index of the chosen neighbor. This is done through sharing of the neighbor identifier among `IloScanNHood`, `IloTest`, and `IloNotify`.

Implementation

The execute method of this goal can be implemented as follows:

```
IlcGoal IlcSingleMoveI::execute(){
  IlcGoal scan = IloScanNHood(solver, nh, nid, solution, mh.getDeltaCheck());
  IlcGoal testGoal = IloTest(solver, mh, nid);
  IlcGoal exploreNHood = IlcAnd(IloStart(solver, mh, solution),
                                scan,
                                testGoal,
                                subgoal,
                                testGoal);
```

```
  IlcGoal saveGoal = IlcAnd(IloNotify(solver, nh, nid),
                            IloNotify(solver, mh, nid),
                            IloStoreSolution(solver, solution));

  return IlcAnd(IlcSelectSearch(exploreNHood, sel), saveGoal);
}
```

See Also: IlcNeighborIdentifier, IloApplyMetaHeuristic, IloMetaHeuristic, IloNeighborIdentifier, IloNHood, IloNotify, IloScanNHood

Global function IlcChooseMinMinFloat

`public IlcInt **IlcChooseMinMinFloat**(const IlcFloatVarArray vars)`

Definition file: ilsolver/linfloat.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the smallest minimum from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcFloatVarArray, IloGenerate

Global function IlcSubset

```
public IlcConstraint IlcSubset(IlcAnySetVar a, IlcAnySetVar b)
public IlcConstraint IlcSubset(IlcAnySet a, IlcAnySetVar b)
public IlcConstraint IlcSubset(IlcAnySetVar a, IlcAnySet b)
public IlcConstraint IlcSubset(IlcIntSetVar a, IlcIntSetVar b)
public IlcConstraint IlcSubset(IlcIntSet a, IlcIntSetVar b)
public IlcConstraint IlcSubset(IlcIntSetVar a, IlcIntSet b)
```

Definition file: ilsolver/ilcset.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a constraint that `a` must be a strict subset of `b`. That is, they cannot be equal, and every element of `a` is also an element of `b`.

See Also: IlcAnySetVar, IlcCard, IlcConstraint, IlcIntSetVar, IlcSubsetEq

Global function IlcExponent

```
public IlcFloatExp IlcExponent(const IlcFloatExp x)
public IlcFloat IlcExponent(IlcFloat x)
```

Definition file: ilsolver/nlinflt.h
Include file: <ilsolver/ilosolver.h>

This function creates a new constrained expression equal to the exponentiation of the constrained expression `x`. The effects of this function are reversible. If its argument is an instance of `IlcFloat`, it simply returns the exponentiation of its argument. With Solver, you should use this function instead of the standard C++ function `exp` in order to conform to the IEEE 754 standard for floating-point calculations.

See Also: IlcFloatExp, IlcMonotonicIncreasingFloatExp, IlcPower, IlcSquare

Global function IloGenerate

```
public IloGoal IloGenerate(const IloEnv env, const IloNumVarArray vars, const
IloChooseIntIndex=IloChooseFirstUnboundInt)
public IloGoal IloGenerate(const IloEnv env, const IloAnyVarArray vars, const
IloChooseAnyIndex=IloChooseFirstUnboundAny)
public IloGoal IloGenerate(const IloEnv env, const IloAnyVarArray vars, const
IloChooseAnyIndex choose, const IloAnyValueSelector select)
public IloGoal IloGenerate(const IloEnv env, const IloNumVarArray vars, const
IloChooseIntIndex choose, const IloIntValueSelector select)
public IloGoal IloGenerate(const IloEnv env, const IloNumSetVarArray vars, const
IloChooseIntSetIndex=IloChooseFirstUnboundIntSet)
public IloGoal IloGenerate(const IloEnv env, const IloAnySetVarArray vars, const
IloChooseAnySetIndex=IloChooseFirstUnboundAnySet)
public IloGoal IloGenerate(const IloEnv env, const IloNumSetVarArray vars, const
IloChooseIntSetIndex choose, const IloIntSetValueSelector select)
public IloGoal IloGenerate(const IloEnv env, const IloAnySetVarArray vars, const
IloChooseAnySetIndex choose, const IloAnySetValueSelector select)
```

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal, using the criterion indicated by `choose` and the selector indicated by `select`. This goal is part of the enumeration algorithm available in an instance of `IloSolver`. It enables you to set parameters for choosing the order in which variables are tried during the search for a solution.

This goal binds each constrained variable in its argument `vars`. It does so by calling the function `IloInstantiate` for each of them. You control the order in which the variables are bound by means of the criterion `choose`.

The goal returned by this function differs from the one returned by `IloBestGenerate`: `IloBestGenerate` calls `IloBestInstantiate`, which tries only one value for each variable, whereas this one calls `IloInstantiate`, which may try all the values in the domain of each variable.

When it takes an instance of the class `IloEnv` as a parameter, it returns an instance of `IloGoal` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the goal that this function returns as an instance of `IlcGoal` for use during a Solver search.

This function works on numerical variables of type `Float` and type `Int`.

See Also: IloBestGenerate, IloGoal, IloInstantiate, IlcGenerate

Global function IloChooseFirstUnboundInt

```
public IlcInt IloChooseFirstUnboundInt(const IlcIntVarArray vars)
```

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the first unbound constrained variable that it encounters in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseIntIndex, IloGenerate

Global function IloReplaceSolutions

```
public IloPoolProc IloReplaceSolutions(IloEnv env, IloSolutionPool target,
IloSolutionPoolSelector selector=0)
```

Definition file: ilsolver/iimproc.h
Include file: <ilsolver/iim.h>

Returns a processor that will replace elements from a supplied pool with elements of the input pool.
This function creates a processor from an environment `env`, a solution pool `target` and a selector of solutions `selector`. The idea of the processor is to replace some of the solutions in `target` by those which are the input of the processor, such as to keep the size of `target` constant. The (optional) selector `selector` decides which solutions should be replaced. If none is specified, a selector selecting the worst solutions is used. The solutions identified by the selector are removed from the `target` and output by the processor.

The processor works as follows:

1. Let *n* be the number of solutions in the input pool.
2. `selector` is asked *n* times to select a solution from target.
3. Each solution selected is removed from `target` and added to the processor's output pool.
4. All solutions in the input pool are added to `target`.

The following code creates a processor which will replace some members of the population by offspring. The choice of population members to remove is made by a tournament selector, with a tournament size of 3. The tournament selector favors worse solutions so that poorer members of the population are replaced. Replaced solutions (dead ones) are supplied to the `IloDestroyAll` processor which will destroy them and reclaim the memory used:

```
IloSolutionPool offspring(env, "offspring");
IloTournamentSelector<IloSolution,IloSolutionPool> deadSelector(
  env, 3, IloWorstSolutionComparator(env)
);
IloGoal replacementGoal = IloExecuteProcessor(env,
  offspring >>
  IloReplaceSolutions(env, population, deadSelector) >>
  IloDestroyAll(env)
);
```

Global function IloSquare

```
public IloNumExprArg IloSquare(const IloNumExprArg arg)
public IloNum IloSquare(IloNum val)
public IloInt IloSquare(IloInt val)
public IloInt IloSquare(int val)
public IloIntExprArg IloSquare(const IloIntExprArg arg)
```

Definition file: ilconcert/iloexpression.h

Returns the square of its argument.
Concert Technology offers predefined functions that return an expression from an algebraic function over expressions. These predefined functions also return a numeric value from an algebraic function over numeric values as well.

`IloSquare` returns the square of its argument (that is, `val*val` or `expr*expr`).

What Is Extracted

`IloSquare` is extracted by an instance of `IloCplex` as a quadratic term. If the quadratic term is positive semi-definite, it may appear in an objective function or constraint.

`IloSquare` is extracted by an instance of `IloCP` or `IloSolver` as an instance of `IlcSquare`.

Global function IloSetValue

`public IloGoal IloSetValue(const IloEnv env, const IloNumVar var, IloNum value)`

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal for the constrained numeric variable `var` and sets `value` as its value.

When it takes an instance of the class `IloEnv` as a parameter, it returns an instance of `IloGoal` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the goal that it returns as an instance of `IlcGoal` for use during a Solver search.

For more information, see `IloNullIntersect`.

This function works on numerical variables of type `Float` and type `Int`.

See Also: IloGoal, IlcSetValue, IloNullIntersect

Global function IloGeLex

`public IloConstraint IloGeLex(IloEnv, IloIntExprArray, IloIntExprArray, const char *=0)`

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

The `IloGeLex` function returns a constraint which maintains two arrays to be lexicographically ordered.

More specifically, `IloGeLex(x, y)` maintains that `x` is greater than or equal to `y` in the lexicographical sense of the term. This mean that either both arrays are equal or that there exists `i < size(x)` such that for all `j < i`, `x[j] = y[j]` and `x[i] > y[i]`.

Note that the size of the two arrays must be the same.

See Also: IloLeLex, IlcLeLex, IlcGeLex

Global function IloRandomPerturbation

```
public IloGoal IloRandomPerturbation(IloEnv env, IloGoal goal, IloNum probability)
public IlcGoal IlcRandomPerturbation(IlcGoal goal, IlcFloat probability)
```

Definition file: ilsolver/iimgoal.h
Include file: <ilsolver/iim.h>

Returns a goal which randomly permutes the search tree branches of another goal.
This function returns a goal which randomly permutes the search tree branches of another goal in the environment `env`. The idea of the perturbation branch selector is to swap, with a given probability `probability` the branches of each `IlcOr` constructed by the goal `goal`. In this way, randomized versions of standard goals can be created, which is a simple way to create randomized populations for evolutionary algorithms. This goal draws random numbers from the generator of the solver on which it is used.

Global function IlcRemoveValue

`public IlcGoal IlcRemoveValue(const IlcIntVar var, const IlcInt val)`

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function returns a goal which removes the value `val` from the variable `var`.

See Also: IlcSetValue

Global function IloComposeLexical

```
public IloLexicographicComparator< IloObject > IloComposeLexical(IloComparator<
IloObject > a, IloComparator< IloObject > b)
public IloLexicographicComparator< IloObject > IloComposeLexical(IloComparator<
IloObject > a, IloComparator< IloObject > b, IloComparator< IloObject > c)
public IloLexicographicComparator< IloObject > IloComposeLexical(IloComparator<
IloObject > a, IloComparator< IloObject > b, IloComparator< IloObject > c,
IloComparator< IloObject > d)
public IloLexicographicComparator< IloObject > IloComposeLexical(IloComparator<
IloObject > a, IloComparator< IloObject > b, IloComparator< IloObject > c,
IloComparator< IloObject > d, IloComparator< IloObject > e)
```

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

Creates a lexicographic composite comparator from existing comparators.
This function creates a lexicographic composite comparator from existing comparators.

For more information, see Selectors.

See Also: IloLexicographicComparator

Global function IloSubset

```
public IloConstraint IloSubset(const IloEnv env, const IloAnySetVar var1, const
IloAnySetVar var2)
public IloConstraint IloSubset(const IloEnv env, const IloAnySet var1, const
IloAnySetVar var2)
public IloConstraint IloSubset(const IloEnv env, const IloAnySetVar var1, const
IloAnySet var2)
public IloConstraint IloSubset(const IloEnv, const IloIntSetVar var1, const
IloIntSetVar var2)
public IloConstraint IloSubset(const IloEnv, const IloIntSet var1, const
IloIntSetVar var2)
public IloConstraint IloSubset(const IloEnv, const IloIntSetVar var1, const
IloIntSet var2)
```

Definition file: ilconcert/iloanyset.h

For IBM® ILOG® Solver: a constraint forcing one set to be strictly a subset of another set.
This function creates and returns a constraint (an instance of `IloConstraint`) for use in a model. That constraint forces `var1` to be strictly a subset of `var2`. That is, there is at least one element of `var2` not in `var1`, and all elements of `var1` are in `var2`.

In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

Global function IloNotify

```
public IloGoal IloNotify(IloEnv env, IloNHood nhood, IloNeighborIdentifier nid)
public IloGoal IloNotify(IloEnv env, IloMetaHeuristic mh)
public IloGoal IloNotify(IloEnv env, IloMetaHeuristic mh, IloNeighborIdentifier
nid)
public IlcGoal IloNotify(IloSolver solver, IloNHood nhood, IlcNeighborIdentifier
nid)
public IlcGoal IloNotify(IloSolver solver, IloMetaHeuristic mh,
IlcNeighborIdentifier nid)
public IlcGoal IloNotify(IloSolver solver, IloMetaHeuristic mh)
```

Definition file: ilsolver/iimls.h
Include file: <ilsolver/iimls.h>

These functions return goals that perform notification to either a metaheuristic or a neighborhood that a move is being taken.

If specified, `nid` is used to communicate the index and the delta of a neighbor supplied by `IloScanDeltas` and `IloScanNHood`.

Conversely, `IloNotify(IloSolver solver, IloMetaHeuristic mh)` returns a goal for the specified solver that calls `mh.notify(solver, IloSolution())` before succeeding.

See Also: IloMetaHeuristic, IloNHood, IloScanDeltas, IloScanNHood, IloSingleMove, IloTest

Global function IlcPack

`public IlcConstraint IlcPack(IlcIntVarArray load, IlcIntVarArray where, IlcIntArray weight, IlcIntVar used)`

Definition file: ilsolver/ilcint.h
Include file: ilsolver/ilosolver.h

The `IlcPack` function returns a constraint which maintains the load of a set of containers, given a set of weighted items and an assignment of items to containers. Consider that we have *n* items and *m* containers. Each item *i* has an integer weight *w[i]* and a constrained integer variable *p[i]* associated with it indicating in which container (numbered contiguously from 0) item *i* is to be placed. No item can be split up, and so can go in only one container. Also associated with each container *j* is an integer variable *l[j]* representing the load in that container; that is, the sum of the weights of the items which have been assigned to that container. A capacity can be set for each container placing an upper bound on this load variable. The constraint also ensures that the total sum of the loads of the containers is equal to the sum of the weights of the items being placed. The number of containers which are used is also maintained, the definition of usage being that at least one item is placed in the container.

Global function IlcPack

`public IlcConstraint **IlcPack**(IlcIntVarArray load, IlcIntVarArray where, IlcIntArray weight, IlcIntSetVar used)`

Definition file: ilsolver/ilcint.h
Include file: ilsolver/ilosolver.h

The `IlcPack` function returns a constraint which maintains the load of a set of containers, given a set of weighted items and an assignment of items to containers. Consider that we have *n* items and *m* containers. Each item *i* has an integer weight *w[i]* and a constrained integer variable *p[i]* associated with it indicating in which container (numbered contiguously from 0) item *i* is to be placed. No item can be split up, and so can go in only one container. Also associated with each container *j* is an integer variable *l[j]* representing the load in that container; that is, the sum of the weights of the items which have been assigned to that container. A capacity can be set for each container placing an upper bound on this load variable. The constraint also ensures that the total sum of the loads of the containers is equal to the sum of the weights of the items being placed. The indices of the set of containers used is also maintained, the definition of usage being that at least one item is placed in the container.

See Also: IlcCard

Global function IlcPack

```
public IlcConstraint IlcPack(IlcIntVarArray load, IlcIntVarArray where, IlcIntArray weight)
```

Definition file: ilsolver/ilcint.h
Include file: ilsolver/ilosolver.h

The `IlcPack` function returns a constraint which maintains the load of a set of containers, given a set of weighted items and an assignment of items to containers. Consider that we have *n* items and *m* containers. Each item *i* has an integer weight *w[i]* and a constrained integer variable *p[i]* associated with it indicating in which container (numbered contiguously from 0) item *i* is to be placed. No item can be split up, and so can go in only one container. Also associated with each container *j* is an integer variable *l[j]* representing the load in that container; that is, the sum of the weights of the items which have been assigned to that container. A capacity can be set for each container placing an upper bound on this load variable. The constraint also ensures that the total sum of the loads of the containers is equal to the sum of the weights of the items being placed.

Global function IloEqMin

```
public IloConstraint IloEqMin(const IloEnv, const IloIntSetVar var1, const
IloIntVar var2, const IloIntToIntFunction f)
public IloConstraint IloEqMin(const IloEnv, const IloIntSetVar var1, const
IloIntVar var2, const IloIntToIntVarFunction f)
```

Definition file: ilconcert/iloset.h

For IBM® ILOG® Solver: a constraint forcing a variable to the minium of returned values.
This function creates and returns a constraint (an instance of IloConstraint) for use in a model. The constraint forces `var2` to be the minimum of the values returned by the function `f` when it is applied to the variable `var1`.

In order for the constraint to take effect, you must add it to a model with the template IloAdd or the member function IloModel::add and extract the model for an algorithm with the member function IloAlgorithm::extract.

Global function IloLimitOperator

```
public IloPoolOperator IloLimitOperator(IloEnv env, IloPoolOperator op,
IloSearchLimit limit, const char * name=0)
```

Definition file: ilsolver/iimoperator.h
Include file: <ilsolver/iim.h>

Limits the execution of a pool operator.
This function creates a new operator which behaves as the passed operator `op` but which is limited in its search by the limit `limit`. `name`, if provided, becomes the name of the newly limited operator.

Global function IloContinue

`public IloNHood IloContinue(IloEnv env, IloNHood nhood, IloInt offset=0, const char * name=0)`

Definition file: ilsolver/iimnhood.h
Include file: <ilsolver/iimls.h>

This function returns a neighborhood that behaves like `nhood`, except that when a neighborhood move is taken (as notified to the neighborhood through `IloNHood::notify`), subsequent scanning begins from index `index + offset + 1`, modulo the size of neighborhood `nhood`.

This type of neighborhood modification can be useful in a first-acceptance scenario where reinvestigating already explored indices is often less fruitful that exploring new indices.

See Also: IloNHood, IloNHoodI

Global function IlcSequence

```
public IlcConstraint IlcSequence(IlcInt nbMin, IlcInt nbMax, IlcInt seqWidth,
IlcIntVarArray vars, IlcIntArray values, IlcIntVarArray cards)
public IlcConstraint IlcSequence(IlcInt nbMin, IlcInt nbMax, IlcInt seqWidth,
IlcIntVarArray vars, IlcIntArray values, IlcIntVarArray cards, IlcFilterLevel
level)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a constraint that works on sequences.

A sequence constraint groups several constraints created by the functions `IlcCard` or `IlcDistribute` into only one constraint in order to reduce domains of constrained variables more effectively.

The parameter `nbMin` indicates a minimum number of allowable values, and `nbMax` indicates a maximum number of allowable values. The parameter `seqWidth` indicates the number of elements in a sequence. The parameter `cards` indicates an array of cardinalities (that is, how many occurrences).

In the new constraint created by this function, the constrained variables in the array `cards` will be equal to the number of occurrences in the array `vars` of the values in the array `values` such that for each sequence of `seqWidth` (a number) consecutive constrained variables of `vars` at least `nbMin` and at most `nbMax` values of `values` are assigned to a constrained variable of the sequence.

The arrays `cards` and `values` must be the same length; otherwise, Solver will throw an exception (an instance of `IloSolver::SolverErrorException`). However, note that the array `vars` can contain values other than those that appear in the array `values`.

If you do not explicitly state a filter level, then Solver uses the default filter level for this constraint. The optional argument `level` can take values of the enumeration `IlcFilterLevel`. Its lowest value is `IlcBasic`. The amount of domain reduction during propagation depends on that value. The value `IlcExtended` causes more domain reduction than does `IlcBasic`; it also takes longer to run. See `IlcFilterLevel` for an explanation of filter levels and their effect on constraint propagation.

See Also: IlcCard, IlcDistribute, IlcFilterLevel

Global function IlcChooseMaxSizeFloat

`public IlcInt **IlcChooseMaxSizeFloat**(const IlcFloatVarArray vars)`

Definition file: ilsolver/linfloat.h
Include file: <ilsolver/ilosolver.h>

This function returns the index (greater than or equal to 0 (zero)) of the unbound constrained variable with the greatest domain from among the constrained variables in the array of constrained variables, `vars`. If all those variables have been bound already, then this function returns -1.

This function is a predefined choice function, one of the criteria that you can use to set parameters for the order in which Solver binds constrained variables during the search for a solution.

See Also: IloBestGenerate, IlcChooseFirstUnboundFloat, IlcFloatVarArray, IloGenerate

Global function IloExponent

```
public IloNumExprArg IloExponent(const IloNumExprArg arg)
public IloNum IloExponent(IloNum val)
```

Definition file: ilconcert/iloexpression.h

Returns the exponent of its argument.
Concert Technology offers predefined functions that return an expression from an algebraic function on expressions. These predefined functions also return a numeric value from an algebraic function on numeric values as well.

`IloExponent` returns the exponentiation of its argument. In order to conform to IEEE 754 standards for floating-point arithmetic, you should use this function in your Concert Technology applications, rather than the standard C++ `exp`.

Global function IloInitializeImpactGoal

```
public IloGoal IloInitializeImpactGoal(IloEnv env, IloInt depth=-1)
```

Definition file: ilsolver/custgoal.h
Include file: <ilsolver/ilosolver.h>

This function returns a goal that initializes impact values by performing a dichotomic search on each variable for which the impact is going to be maintained (given as a parameter of the function `IlcImpactInformation`). The maximum depth of the diochotomic search is `depth`. A larger depth provides more accurate impacts, but the goal needs more time to compute these impacts. When `depth` is equal to -1, there is no depth limit.

See Also: IlcImpactInformation

Global function IlcAllNullIntersect

```
public IlcConstraint IlcAllNullIntersect(IlcIntSetVarArray array)
public IlcConstraint IlcAllNullIntersect(IlcAnySetVarArray array, IlcFilterLevel
level)
public IlcConstraint IlcAllNullIntersect(IlcAnySetVarArray array)
public IlcConstraint IlcAllNullIntersect(IlcIntSetVarArray array, IlcFilterLevel
level)
```

Definition file: ilsolver/ilcset.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a constraint. This constraint insures that the intersection is empty between any two constrained set variables in the array of its argument. In other words, this function extends `IlcNullIntersect` to *arrays* of constrained set variables. This function is for use during Solver search, for example, inside a goal (an instance of `IlcGoal`) or inside a constraint (an instance of `IlcConstraint`). If you are looking for similar functionality for use in a Concert Technology *model*, consider `IloAllNullIntersect`.

When you create a constraint, it has no effect until you post it, for example, by adding it to an instance of `IloSolver`.

If you do not explicitly state a filter level, then Solver uses the default filter level for this constraint (that is, `IlcLow`). The optional argument `level` can take one of three values: `IlcLow`, `IlcBasic`, `IlcExtended`. Domain reduction during propagation depends on the value of `level`. See `IlcFilterLevel` for an explanation of filter levels and their effect on constraint propagation.

IlcLow (default)

With this choice the statement

```
IlcAllNullIntersect(IlcIntSetVarArray array,IlcLow)
```

is more efficient, but leads to the same domain reduction as the following code:

```
IlcInt size=array.getSize();
for(IlcInt i=0;i<size-1;i++){
   for(IlcInt j=i+1;j<size;j++){
        m.add(IlcNullIntersect(array[i],array[j]);
   }
}
```

IlcBasic

This choice causes more domain reduction than `IlcLow`; it also takes longer to run.

IlcExtended

This choice causes more domain reduction than `IlcBasic`; it also takes longer to run.

For more information, see `IloNullIntersect`.

See Also: IlcAnySetVarArray, IlcCard, IlcIntSetVarArray, IlcNullIntersect, IlcFilterLevel, IlcPartition, IloAllNullIntersect

Global function IlcRandomValueEvaluator

```
public IloEvaluator< IlcInt > IlcRandomValueEvaluator(IloEnv env)
public IloEvaluator< IlcInt > IlcRandomValueEvaluator(IloSolver solver)
```

Definition file: ilsolver/custgoal.h
Include file: <ilsolver/ilosolver.h>

This function returns an instance of the evaluator `IloEvaluator<IlcInt>`. The evaluation returns a random value between 0 and 1.

Global function IloNotMember

```
public IloConstraint IloNotMember(const IloEnv env, const IloAnyVar var1, const
IloAnySetVar var2)
```

Definition file: ilconcert/iloanyset.h

show solver

Global function IloNotMember

```
public IloConstraint IloNotMember(const IloEnv env, IloAny val, const IloAnySetVar var2)
```

Definition file: ilconcert/iloanyset.h

show solver

Global function IloNotMember

`public IloConstraint IloNotMember(const IloEnv, const IloNumExprArg expr, const IloNumArray elements)`

Definition file: ilconcert/ilomodel.h

For constraint programming: creates and returns a constraint forcing `expr` not to be a member of `elements`. This function creates and returns a constraint (an instance of `IloConstraint`) for use in a model. The constraint forces `expr` not to be a member of `elements`.

In order for the constraint to take effect, you must add it to a model with the template `IloAdd` or the member function `IloModel::add` and extract the model for an algorithm with the member function `IloAlgorithm::extract`.

Global function IloApply

```
public IloGoal IloApply(const IloEnv env, const IloGoal goal, const
IloBranchSelector branchSelector)
```

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function returns a goal that applies the branch selector `s` to the search tree defined by the goal `g`. As the goal handler explores the search tree, each choice point is controlled by the branch selector.

It takes an instance of the class `IloEnv` as a parameter, it returns an instance of `IloGoal` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the goal that it returns as an instance of `IlcGoal` for use during a Solver search.

See Also: IlcBranchSelector

Global function IloApply

`public IloGoal` **`IloApply`**`(const IloEnv env, const IloGoal g, const IloNodeEvaluator e)`

Definition file: ilsolver/ilosolverint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal for use in a search. This goal applies the evaluator `e` to the search tree defined by the goal `g`. In doing so, it changes the order of evaluation according to `e` of the open nodes of the search tree defined by `g`.

When it takes an instance of the class `IloEnv` (documented in the *Concert Technology Reference Manual*) as a parameter, it returns an instance of `IloGoal` for use with the member functions `IloSolver::startNewSearch` and `IloSolver::solve`. An instance of `IloSolver` extracts the goal that it returns as an instance of `IlcGoal` for use during a Solver search.

See Also: IlcGoal, IloGoal, IloSolver, IlcApply

Global function IlcInstantiate

```
public IlcGoal IlcInstantiate(const IlcIntVar var)
public IlcGoal IlcInstantiate(const IlcAnyVar var)
public IlcGoal IlcInstantiate(const IlcAnyVar var, IlcAnySelect select)
public IlcGoal IlcInstantiate(const IlcIntVar var, IlcIntSelect select)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal, a primitive in the algorithms that search for solutions. The goal `IlcInstantiate` assigns a value to a constrained variable. It uses choice points so that if a failure occurs as a result of that reversible assignment, another value will be assigned to the constrained variable so that the search can continue.

If `var` has already been bound, then `IlcInstantiate` does nothing and succeeds. Otherwise, `IlcInstantiate` sets a choice point, then assigns a value to the constrained variable. In case of failure, the tried-and-failed value is removed from the domain of the constrained variable, and another value not yet used is tried until a value assignment succeeds or the domain is exhausted. In that latter case, the domain becomes empty, and failure occurs. The values are selected by the `select` object, if it is provided. Otherwise, values are tried by default in ascending order.

Implementation

Here's how we could define that goal for `IlcIntVar`, using the constraints `==` and `!=` directly as goals themselves.

```
static ILCGOAL2(IlcIntInstantiate,
                IlcIntVar, var,
                IlcIntSelectI*, select) {
    if (var.isBound()) return 0;
    IlcInt val = (select)? select->select(var) : var.getMin();
    return IlcOr(var == val, IlcAnd(var != val,
                                    this));
}
IlcGoal IlcInstantiate(IlcIntVar var){
    return IlcIntInstantiate(var, 0);
}
IlcGoal IlcInstantiate(IlcIntVar var, IlcIntSelect sel){
    return IlcIntInstantiate(var, sel.getImpl());
}
```

For more information, see the concept Choice Point.

See Also: IlcAnySelect, IlcBestInstantiate, IlcDichotomize, IlcGenerate, IlcGoal, IlcIntSelect

Global function IlcInstantiate

```
public IlcGoal IlcInstantiate(const IlcIntSetVar var)
public IlcGoal IlcInstantiate(const IlcAnySetVar var)
public IlcGoal IlcInstantiate(const IlcAnySetVar var, IlcAnySetSelect select)
public IlcGoal IlcInstantiate(const IlcIntSetVar var, IlcIntSetSelect select)
```

Definition file: ilsolver/intset.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal, a primitive in the algorithms that search for solutions. The goal `IlcInstantiate` assigns a value to a constrained variable. It uses choice points so that if a failure occurs as a result of that reversible assignment, another value will be assigned to the constrained variable so that the search can continue. Its behavior varies slightly, depending on the class of its arguments.

If `var` has already been bound, `IlcInstantiate` does nothing and succeeds. Otherwise, `IlcInstantiate` sets a choice point, then adds an element of the *possible* set to the *required* set of `var`. The added element is chosen by `select`, if it is provided. If `select` has not been provided, the values are tried in ascending order when the values are integers. If failure occurs, the element is removed from the possible set of `var`, and another element is tried.

For more information, see the concept Choice Point.

See Also: IlcAnySetSelect, IlcAnySetVar, IlcBestGenerate, IlcBestInstantiate, IlcGenerate, IlcIntSetSelect, IlcIntSetVar

Global function IlcInstantiate

```
public IlcGoal IlcInstantiate(const IlcFloatVar var, IlcBool
increaseMinFirst=IlcTrue, IlcFloat prec=0)
```

Definition file: ilsolver/nlinflt.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns a goal, a primitive in the algorithms that search for solutions. Its behavior varies slightly, depending on the class of its arguments.

The algorithm that `IlcInstantiate` uses for constrained integer variables and constrained enumerated variables tries every value in the domain. Since the number of elements in the domain of a constrained floating-point variable is very high (typically millions), that approach would not be very efficient. Consequently, `IlcInstantiate` handles the domain of a constrained floating-point variable differently. The idea is to recursively split the domain of the constrained floating-point variable into two parts.

If `var` is already bound, then `IlcInstantiate` does nothing and succeeds. Otherwise, `IlcInstantiate` sets a choice point, then replaces the *domain* of `var` by one of its *halves*, and calls itself recursively. The function stops when the variable is bound or when it is known with a precision smaller than `prec`. If a failure occurs, then the domain is replaced by the other half, and `IlcInstantiate` is called recursively.

The optional argument `increaseMin` should be a Boolean value, either `IlcTrue` or `IlcFalse`. If `increaseMinFirst` is `IlcTrue`, then the upper half domain is tried first; otherwise, the lower half is tried first.

This algorithm is available explicitly for constrained integer variables as the function `IlcDichotomize`.

For more information, see the concept Choice Point.

See Also: IlcBestInstantiate, IlcDichotomize, IlcGenerate, IlcGoal

Global function IlcBranchImpactVarEvaluator

```
public IloEvaluator< IlcIntVar > IlcBranchImpactVarEvaluator(IloEnv env)
public IloEvaluator< IlcIntVar > IlcBranchImpactVarEvaluator(IloSolver solver)
```

Definition file: ilsolver/custgoal.h
Include file: <ilsolver/ilosolver.h>

This function returns an instance of the evaluator `IloEvaluator<IlcIntVar>`. The evaluation returns the result of the function `solver.getBranchImpact(x)`, where `x` is the evaluated variable.

Global function IlcBoolAbstraction

```
public IlcBoolVarArray IlcBoolAbstraction(IlcIntVarArray vars, IlcIntArray values)
public IlcBoolVarArray IlcBoolAbstraction(IlcAnyVarArray vars, IlcAnyArray values)
```

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

This function creates and returns an array of constrained Boolean variables for using during a Solver search. The argument `vars` should be an array of constrained variables. The argument `values` is an array of integers or pointers.

For each `vars[i]`, Solver creates the *Boolean abstraction* of `vars[i]` with respect to `values`. In other words, for every variable in `vars`, `vars[i]`, Solver creates a constrained Boolean variable `w[i]` such that `w[i]=0` if and only if `vars[i]` cannot be bound with a value of `values` and `w[i]=1` if and only if `vars[i]` will necessarily be bound with a value of `values`. Then Solver insures that these properties hold after the definition of the Boolean abstraction.

For a function that returns a constraint (rather than an array), see `IlcEqBoolAbstraction`.

For a constraint suitable for use in a *model*, see `IloBoolAbstraction`.

See Also: IlcAbstraction, IlcEqBoolAbstraction, IloBoolAbstraction

Global function operator||

```
public IloPoolOperator operator||(IloPoolOperator op1, IloPoolOperator op2)
```

Definition file: ilsolver/iimoperator.h
Include file: <ilsolver/iim.h>

Produces the disjunction of two operators.
This disjunction of operators is an operator which, when invoked, invokes `op1`. If successful, the operator succeeds. Otherwise, the disjunction invokes `op2`. If `op2` succeeds, the disjunction succeeds, otherwise it fails.

Note
If no prototype is set on the disjunction, then its prototype is defined to be one from either `op1` or `op2`. The choice is dynamic and depends on whether `op1` or `op2` produced the solution.

Global function operator||

```
public IlcConstraint operator||(const IlcConstraint ct1, const IlcConstraint ct2)
```

Definition file: ilsolver/numi.h
Include file: <ilsolver/ilosolver.h>

This operator creates and returns a constraint: the disjunction of its arguments.

When you create a constraint, it has no effect until you post it.

See Also: IlcConstraint

Global function operator||

```
public IloOr operator||(const IloConstraint constraint1, const IloConstraint
constraint2)
```

Definition file: ilconcert/ilomodel.h

Overloaded C++ operator for a disjunctive constraint.
This overloaded C++ operator creates a disjunctive constraint that represents the disjunction of its two arguments. The constraint can represent a disjunction of two constraints; of a constraint and another disjunction; or of two disjunctions. In order to be taken into account, this constraint must be added to a model and extracted by an algorithm, such as `IloCplex` or `IloSolver`.

Global function operator||

```
public IloPredicate< IloObject > operator||(IloPredicate< IloObject > left,
IloPredicate< IloObject > right)
```

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

Creates a predicate performing OR on two predicates.
This operator creates a new `IloPredicate<IloObject>` instance from two `IloPredicate<IloObject>` instances. The semantics of the combination of the component predicates is that of logical `OR`. That is, the combined predicate will return `IloTrue` for a particular object if and only if one of the component predicates return `IloTrue` for that object.

For more information, see Selectors.

Global function operator*

```
public IloNHood operator*(IloNHood nhood1, IloNHood nhood2)
```

Definition file: ilsolver/iimnhood.h
Include file: <ilsolver/ilosolver.h>

This operator creates a *composed neighborhood*. The neighborhood formed is the “product” of `nhood1` and `nhood2`. See `IloCompose` for full documentation of this operator.

See Also: IloCompose, IloNHood, operator+

Global function operator*

```
public IlcFloatExp operator*(const IlcFloatExp exp1, IlcFloat exp2)
public IlcIntExp operator*(const IlcIntExp exp1, IlcInt exp2)
public IlcIntExp operator*(IlcInt exp1, const IlcIntExp exp2)
public IlcIntExp operator*(const IlcIntExp exp1, const IlcIntExp exp2)
public IlcFloatExp operator*(IlcFloat exp1, const IlcFloatExp exp2)
public IlcFloatExp operator*(const IlcFloatExp exp1, const IlcFloatExp exp2)
public IlcIntToIntStepFunction operator*(const IlcIntToIntStepFunction & f1, IlcInt
k)
public IlcIntToIntStepFunction operator*(IlcInt k, const IlcIntToIntStepFunction &
f1)
```

Definition file: ilsolver/linfloat.h
Include file: <ilsolver/ilosolver.h>

This arithmetic operator multiplies its arguments. It has been overloaded to handle constrained expressions appropriately. The domain of the resulting expression is computed from the domains of the combined expressions as you would expect.

Example

The following code from the example `sendmory.cpp` in the standard distribution creates several constrained expressions.

```
IloSolver s;
IlcIntVar S(s, 1, 9, "S"), E(s, 0, 9, "E"), N(s, 0, 9, "N"),
          D(s, 0, 9, "D"), M(s, 1, 9, "M"), O(s, 0, 9, "O"),
          R(s, 0, 9, "R"), Y(s, 0, 9, "Y");
IlcIntExp send =            1000*S + 100*E + 10*N + D;
IlcIntExp more =            1000*M + 100*O + 10*R + E;
IlcIntExp money = 10000*M + 1000*O + 100*N + 10*E + Y;
```

See Also: IlcFloatExp, IlcIntExp, IlcIntToIntStepFunction

Global function operator*

```
public IloEvaluator< IloObject > operator*(IloEvaluator< IloObject > left, IloEvaluator< IloObject > right)
public IloEvaluator< IloObject > operator*(IloEvaluator< IloObject > left, IloNum c)
public IloEvaluator< IloObject > operator*(IloNum c, IloEvaluator< IloObject > right)
```

Definition file: ilsolver/iloselector.h
Include file: <ilsolver/iloselector.h>

These operators create a composite `IloEvaluator<IloObject>` instance. The semantics of the new evaluator are the multiplication of the values of the component evaluators. The first function combines two evaluators, multiplying their values to generate the combined evaluation. The other two signatures multiply the value returned by the evaluator with an `IloNum` value.

For more information, see Selectors.

Global function operator*

```
public IloNumLinExprTerm operator*(const IloNumVar x, IloInt num)
public IloNumLinExprTerm operator*(IloInt num, const IloNumVar x)
public IloNumLinExprTerm operator*(const IloIntVar x, IloNum num)
public IloNumLinExprTerm operator*(IloNum num, const IloIntVar x)
public IloIntLinExprTerm operator*(const IloIntVar x, IloInt num)
public IloNumExprArg operator*(const IloNumExprArg x, const IloNumExprArg y)
public IloNumExprArg operator*(const IloNumExprArg x, IloNum y)
public IloNumExprArg operator*(IloNum x, const IloNumExprArg y)
public IloIntExprArg operator*(const IloIntExprArg x, const IloIntExprArg y)
public IloIntExprArg operator*(const IloIntExprArg x, IloInt y)
```

Definition file: ilconcert/iloexpression.h

Returns an expression equal to the product of its arguments.
This overloaded C++ operator returns an expression equal to the product of its arguments. Its arguments may be numeric values, numeric variables, or other expressions.

Global function operator*

```
public IloNumToNumStepFunction operator*(const IloNumToNumStepFunction f1, IloNum
k)
public IloNumToNumStepFunction operator*(IloNum k, const IloNumToNumStepFunction
f1)
```

Definition file: ilconcert/ilonumfunc.h

Creates and returns a function equal to its argument function multiplied by a given factor.
These operators create and return a function equal to the function `f1` multiplied by a factor `k` everywhere on the definition interval. The resulting function is defined on the same interval as the argument function `f1`. See also: `IloNumToNumStepFunction`.

Global function IloSimulatedAnnealing

```
public IloMetaHeuristic IloSimulatedAnnealing(IloEnv env, IloRandom rand, IloNum
startTemperature, IloNum reductionFactor, IloNum cutoffProbability=1e-5, IloNum
freezingTemperature=0.0)
public IloMetaHeuristic IloSimulatedAnnealing(IloEnv env, IloNum & temperature,
IloRandom rand, IloNum startTemperature, IloNum reductionFactor, IloNum
cutoffProbability=1e-5, IloNum freezingTemperature=0.0)
```

Definition file: ilsolver/iimmeta.h
Include file: <ilsolver/iimls.h>

This function returns a metaheuristic that implements a simulated annealing behavior. This metaheuristic discards neighbors using a probabilistic rule based on the cost of the neighbor and a notion of *temperature*. We will describe the operation of this metaheuristic as if it is performing a minimization procedure, but maximization can also be performed. Given a current solution of cost *c*, a neighbor of cost *c+d*, a temperature of *T*, and a random number *r* in the range *[0..1)*, then the neighbor is accepted under the following condition:

r <= exp(-d / T)

When the increase in cost *d* is not strictly positive, the move is never rejected. When the increase in cost provided by the move is positive, the move is rejected probabilistically. Larger increases in cost and lower temperatures both make rejection of the move more likely. The current temperature is maintained internally by the returned metaheuristic or stored in the variable `temperature`, so that it can be inspected if desired.

The random number generator `rand` is used to generate the numbers *r* for the rule above. A non-reversible random number generator is recommended, as a reversible one generates the same random number for each leaf tested. To avoid the problem of the different state of the random number generator during re-computation, the simulated annealing test above is only performed when in the original computation mode. No neighbors are rejected when recomputing.

The overall idea of simulated annealing is to begin search at a high temperature to allow degrading moves to be taken readily— which allows the search to wander. Over time, the temperature is reduced, causing the search to be less inclined to accept degrading moves. The parameter `startTemperature` defines the temperature at which to begin the simulated annealing metaheuristic. Each time that the `notify` method is called for the returned metaheuristic, the temperature is reduced by the following rule.

```
currentTemperature = currentTemperature * reductionFactor
```

where `reductionFactor` can range from 0 to 1. Usually, reduction factors are greater than 0.99. A high reduction factor results in slower cooling and, usually, in a better final solution. However, the search takes longer to stabilize in this condition.

If the parameter `cutoffProbability` is specified, it can be used as an efficiency measure. If the chance of accepting a neighbor according to the above probability equation is less than `cutoffProbability`, then it is rejected. This is done at each move by setting the upper bound on the cost equal to:

*c - T * ln (*`cutoffProbability`*)*

where *c* is the current cost and *T* is the current temperature. This allows pruning of neighbors with cost greater than this value without using the probabilistic rule. A cutoff probability of 0 results in a “pure' simulated annealing algorithm.

The parameter `freezingTemperature` can be specified to define a lower limit on the temperature reduction. Calling `IloMetaHeuristic::complete` on the returned metaheuristic returns `IloTrue` if and only if this temperature has been met.

Note

If you are using a neighborhood with the simulated annealing metaheuristic, it is advisable to randomize the neighborhood via `IloRandomize`, otherwise there will be a bias for accepting moves from the beginning of the neighborhood. In addition, it is also advisable to select the first move accepted by the simulated annealing metaheuristic, as this most faithfully follows the original simulated annealing algorithm.

For more information, see `IloRandom` in the *Concert Technology Reference Manual*.

See Also: IloFirstSolution, IloMetaHeuristic, IloRandomize

Global function IloStart

```
public IlcGoal IloStart(IloSolver solver, IloMetaHeuristic mh, IloSolution
solution)
public IloGoal IloStart(IloEnv env, IloNHood nhood, IloSolution solution)
public IloGoal IloStart(IloEnv env, IloMetaHeuristic mh, IloSolution solution)
public IlcGoal IloStart(IloSolver solver, IloNHood nhood, IloSolution solution)
```

Definition file: ilsolver/iimls.h
Include file: <ilsolver/iimls.h>

These functions return goals that start either a metaheuristic or a neighborhood.

When `mh` is passed as a parameter, these functions return a goal which calls `mh.start(solver, solution)`. If this call does not cause a failure and returns `IloTrue`, the goal succeeds. Otherwise it fails.

When `nhood` is passed as a parameter, these functions return a goal which calls `nhood.start(solver, solution)` before succeeding. You should not normally need to use this goal as `IloScanNHood` automatically performs this action.

Use of the `IloSingleMove` goal renders the use of both forms of `IloStart` unnecessary.

See Also: IloStart, IloApplyMetaHeuristic, IloMetaHeuristic, IloNHood, IloNotify, IloScanNHood, IloSingleMove, IloTest

Macro ILCANYPREDICATE0

Definition file: ilsolver/ilcany.h

```
ILCANYPREDICATE0(name)
ILCANYPREDICATE1(name, type1, nameArg1)
ILCANYPREDICATE2(name, type1, nameArg1, type2, nameArg2)
ILCANYPREDICATE3(name, type1, nameArg1, type2, nameArg2, type3, nameArg3)
ILCANYPREDICATE4(name, type1, nameArg1, type2, nameArg2, type3, nameArg3, type4,
nameArg4)
ILCANYPREDICATE5(name, type1, nameArg1, type2, nameArg2, type3, nameArg3, type4,
nameArg4, type5, nameArg5)
ILCANYPREDICATE6(name, type1, nameArg1, type2, nameArg2, type3, nameArg3, type4,
nameArg4, type5, nameArg5, type6, nameArg6)
```

This macro defines a predicate class named `nameI` with *n* data members. When *n* is greater than 0, the types and names of the data members must be supplied as arguments to the macro. Each data member is defined by its type `T`*i* and a name `data`*i*. The call to the macro must be followed immediately by the body of the `isTrue` member function of the predicate class being defined. Besides the definition of the class `nameI`, this macro also defines a function named `name` that creates an instance of the class `nameI` and that returns an instance of the class `IlcAnyPredicate` that points to it.

Solver does not check the arity of the predicate that you defined. It assumes that the size of the array (an instance of `IlcAnyArray`) passed as an argument to the member function `IlcAnyPredicate::isTrue` will always be the same. It also assumes that the name of the array passed as an argument is `val`. That is, you *must* use that name to define a predicate.

You are not obliged to use this macro to define predicates on arbitrary objects. When the macro seems too restrictive for your purposes, we recommend that you define a predicate class directly by subclassing `IlcAnyPredicateI`.

Since the argument `name` is used to name the predicate class, it is not possible to use the same name for several predicate definitions.

For an example of how to use a similar macro, see the macro `ILCINTPREDICATE0`.

See Also: IlcAnyArray, ILCANYPREDICATE0, ILCINTPREDICATE0, IlcTableConstraint

Macro ILCARRAY

Definition file: ilsolver/basic.h

ILCARRAY(t)

This macro defines one-dimensional arrays for a given *type* of object, where `type` must be the name of a handle class. The macro creates an implementation class, named `typeArrayI`, and a handle class, named `typeArray`. (You should replace `type` by the actual name of the original handle class.) Instances of these classes are arrays of elements of the given handle class `type`.

Example

This statement

```
ILCARRAY(IlcIntervalActivity);
```

creates the handle class `IlcIntervalActivityArray` and the implementation class `IlcIntervalActivityArrayI.`

This macro could be implemented like this:

```
#define ILCARRAY(type)
 ILCARRAY2(name2(type,Array),type)
```

See Also: ILCARRAY2, IlcChooseAnyIndex1, IlcChooseAnyIndex2

Macro ILCARRAY2

Definition file: ilsolver/basic.h

ILCARRAY2(name, t)

This macro defines a handle class for one-dimensional arrays of a given `type` of object, where `type` may be either the name of a handle class or the name of a class followed by *. The macro creates an implementation class, *name*`I`, and a handle class, *name*. (You should replace `name` by the identifier you want to use for the new class.) Instances of this new class are arrays of elements of the given `type`.

Example

This statement

```
ILCARRAY2(MyClassArray, MyClass*);
```

creates a handle class `MyClassArray` of pointers to objects of the class `MyClass`.

See Also: ILCARRAY, IlcChooseAnyIndex1, IlcChooseAnyIndex2

Macro IlcChooseAnyIndex1

Definition file: ilsolver/basic.h

IlcChooseAnyIndex1(name, t, var, condition, criterion)

This macro defines a new choice function (a criterion) in Solver for setting parameters on the search for a solution; you use this macro if you have *one* integer criterion. (The “Any” in its name refers to the fact that you can use any `type` as long as `type` indicates a handle class.)

This macro defines a choice function for objects of the type indicated by `type`. The name of the function will be `name`. This function will take an array (an instance of `typeArray`) of elements (instances of `type`). The argument, `criterion`, should be a C++ expression of type `IlcInt`. In that expression, the object to evaluate *must be denoted by*`var`. The function named `name` returns the index of the object of the type indicated by `type` from among those objects for which `condition` is `IlcTrue` and that minimizes the expression `criterion`. If `condition` is `IlcFalse` for all objects, then this function returns -1.

See Also: ILCARRAY, ILCARRAY2, IlcChooseAnyIndex2

Macro IlcChooseAnyIndex2

Definition file: ilsolver/basic.h

`IlcChooseAnyIndex2``(name, t, var, condition, criterion1, criterion2)`

This macro defines a new choice function (a criterion) in Solver for setting parameters on the search for a solution; you use this macro if you have *two* integer criteria. (The “Any” in its name refers to the fact that you can use any `type` as long as `type` indicates a handle class.)

This macro defines a choice function for objects of the type indicated by `type`. The name of the function will be `name`. This function will take an array (an instance of `typeArray`) of elements (instances of `type`). The argument, `criterion`, should be a C++ expression of type `IlcInt`. In that expression, the object to evaluate *must be denoted by*`var`. The function named `name` returns the index of the object of the type indicated by `type` from among those objects for which `condition` is `IlcTrue` and that minimizes the expressions `criterion1` and `criterion2`. If more than one object satisfies `condition` and minimizes `criterion1`, then `criterion2` will be used to distinguish between them. If `condition` is `IlcFalse` for all objects, then this function returns -1.

See Also: ILCARRAY, ILCARRAY2, IlcChooseAnyIndex1

Macro IlcChooseFloatIndex1

Definition file: ilsolver/critmac.h

IlcChooseFloatIndex1`(name, criterion, type)`

This macro defines a new choice function (a criterion) in Solver for setting parameters on the search for a solution; you use this macro if you have *one* floating-point criterion.

This macro defines a choice function for constrained variables of type `varType`. The name of the function will be `name`. The second argument, `criterion`, should be a C++ expression of type `IlcFloat`. In that expression, the constrained variable to evaluate *must be denoted by*`var`. The index of the variable in the array is `varIndex`. The function named `name` returns the index of the constrained variable of type `varType` that minimizes the expression `criterion`. If all the constrained variables have already been bound, then this function returns -1.

Example

As an example of how to use that macro, the predefined criteria for constrained floating-point variables could be defined in the following way:

```
IlcChooseFloatIndex1
  (IlcChooseMinSizeFloat, var.getSize(), IlcFloatVar)
IlcChooseFloatIndex1
  (IlcChooseMaxSizeFloat,-var.getSize(), IlcFloatVar)
IlcChooseFloatIndex1(IlcChooseMinMinFloat, var.getMin(), IlcFloatVar)
IlcChooseFloatIndex1(IlcChooseMinMaxFloat, var.getMax(), IlcFloatVar)
IlcChooseFloatIndex1
  (IlcChooseMaxMinFloat, -var.getMin(), IlcFloatVar)
IlcChooseFloatIndex1
  (IlcChooseMaxMaxFloat, -var.getMax(), IlcFloatVar)
```

See Also: IlcChooseFirstUnboundFloat, IlcChooseFloatIndex, IlcChooseMaxMaxFloat, IlcChooseMaxMinFloat, IlcChooseMaxSizeFloat, IlcChooseMinMaxFloat, IlcChooseMinMinFloat, IlcChooseMinSizeFloat

Macro IlcChooseFloatIndex2

Definition file: ilsolver/critmac.h

`IlcChooseFloatIndex2``(name, criterion1, criterion2, type)`

This macro defines a new choice function (a criterion) in Solver for setting parameters on the search for a solution; you use this macro if you have *two* floating-point criteria.

This macro defines a choice function for constrained variables of type `varType`. The name of the function will be `name`. The second and third arguments, `criterion1` and `criterion2`, should be C++ expressions of type `IlcFloat`. In these expressions, the constrained variable to evaluate *must be denoted by*`var`. The index of the variable in the array is `varIndex`. The function named `name` returns the index of the constrained variable of type `varType` that minimizes the expressions `criterion1` and `criterion2`. If more than one constrained variable minimizes the first criterion, then the second criterion will be used to distinguish between them. If all the constrained variables have already been bound, then this function returns -1.

See Also: IlcChooseFirstUnboundFloat, IlcChooseFloatIndex, IlcChooseMaxMaxFloat, IlcChooseMaxMinFloat, IlcChooseMaxSizeFloat, IlcChooseMinMaxFloat, IlcChooseMinMinFloat, IlcChooseMinSizeFloat

Macro IlcChooseIndex1

Definition file: ilsolver/critmac.h

`IlcChooseIndex1`

This macro defines a new choice function (a criterion) in Solver for setting parameters on the search for a solution, you use this macro if you have *one* integer criterion.

This macro defines a choice function for constrained variables of type `varType`. The name of the function will be `name`. The second argument, `criterion`, should be a C++ expression of type `IlcInt`. In that expression, the constrained variable to evaluate must be denoted by `var`. The index of the variable in the array is `varIndex`. The function named `name` returns the index of the constrained variable of type `varType` that minimizes the expression `criterion`. If all the constrained variables have already been bound, then this function returns -1.

Example

As an example of how to use IlcChooseIndex1, the predefined criteria for constrained integer variables could be defined in the following way with this macro:

```
IlcChooseIndex1(IlcChooseMinSizeInt,  var.getSize(), IlcIntVar);
IlcChooseIndex1(IlcChooseMaxSizeInt, -var.getSize(), IlcIntVar);
IlcChooseIndex1(IlcChooseMinMinInt,   var.getMin(),  IlcIntVar);
IlcChooseIndex1(IlcChooseMinMaxInt,   var.getMax(),  IlcIntVar);
IlcChooseIndex1(IlcChooseMaxMinInt,  -var.getMin(),  IlcIntVar);
IlcChooseIndex1(IlcChooseMaxMaxInt,  -var.getMax(),  IlcIntVar);
```

Other classes of variables can also be used:

```
IlcChooseIndex1(IlcChooseMinSizeAny,  var.getSize(), IlcAnyVar);
```

Implementation

Here's how this macro might be implemented.

```
#define IlcChooseIndex1(name, criterion, type)
 IlcInt       name (IlcInt theIlcChooseIndexSize,
           name2(type,I**) vars) {
     IlcInt varIndex;
     type var;
     name2(type,I**) tmp=vars;
     IlcInt indexBest=-1;
     IlcInt value, min = IlcIntMax;
     for (varIndex=0;
        varIndex<theIlcChooseIndexSize;
        varIndex++, tmp++) {
      var = type(*tmp);
      if (!var.isBound()) {
          value = criterion;
          if (min > value) {
              indexBest = varIndex;
              min = value;
          }
      }
     }
     return indexBest;
 }
 IlcInt name (const name2(type,Array) array){
    return name (array.getSize(), array.getImpl()->getArray());
 }
```

See Also: IlcChooseFirstUnboundInt, IlcChooseIndex2, IlcChooseIntIndex, IlcChooseMaxMaxInt, IlcChooseMaxMinInt, IlcChooseMaxSizeInt, IlcChooseMinMaxInt, IlcChooseMinMinInt, IlcChooseMinSizeInt

Macro IlcChooseIndex2

Definition file: ilsolver/critmac.h

IlcChooseIndex2

This macro defines a new choice function (a criterion) in Solver for setting parameters on the search for a solution; you use this macro if you have *two* integer criteria.

This macro defines a choice function for constrained variables of type `varType`. The name of the function will be `name`. The second and third arguments, `criterion1` and `criterion2`, should be C++ expressions of type `IlcInt`. In these expressions, the constrained variable to evaluate *must be denoted by*`var`. The index of the variable in the array is `varIndex`. The function named `name` returns the index of the constrained variable of type `varType` that minimizes the expressions `criterion1` and `criterion2`. If more than one constrained variable minimizes the first criterion, then the second criterion will be used to distinguish between them. If all the constrained variables have already been bound, then this function returns -1.

Implementation

Here's how this macro might be implemented.

```
#define IlcChooseIndex2(name, criterion1, criterion2, type)
 IlcInt name (IlcInt theIlcChooseIndexSize,
           name2(type, I**) vars) {
     IlcInt varIndex;
     type var;
     name2(type,I**) tmp=vars;
     IlcInt indexBest = -1;
     IlcInt value1, min1 = IlcIntMax;
     IlcInt value2, min2 = IlcIntMax;
     for (varIndex=0; varIndex < theIlcChooseIndexSize; varIndex++, tmp++)
 {
      var = type(*tmp);
      if (!var.isBound()) {
          value1 = criterion1;
          if (value1 < min1) {
              min1 = value1;
              indexBest = varIndex;
              min2 = criterion2;
          }
          else {
              if (value1 == min1) {
                  value2 = criterion2;
                  if (value2 < min2) {
                      min2 = value2;
                      indexBest = varIndex;
                  }
              }
          }
      }
     }
     return indexBest;
 }
 IlcInt name (const name2(type,Array) array){
     return name (array.getSize(), array.getImpl()->getArray());
 }
```

See Also: IlcChooseFirstUnboundInt, IlcChooseIndex1, IlcChooseIntIndex, IlcChooseMaxMaxInt, IlcChooseMaxMinInt, IlcChooseMaxSizeInt, IlcChooseMinMaxInt, IlcChooseMinMinInt, IlcChooseMinSizeInt

Macro ILCCTDEMON0

Definition file: ilsolver/basic.h

```
ILCCTDEMON0(name, IlcCtClass, IlcFnName)
ILCCTDEMON1(name, IlcCtClass, IlcFnName, t1, nA1)
ILCCTDEMON2(name, IlcCtClass, IlcFnName, t1, nA1, t2, nA2)
ILCCTDEMON3(name, IlcCtClass, IlcFnName, t1, nA1, t2, nA2, t3, nA3)
ILCCTDEMON4(name, IlcCtClass, IlcFnName, t1, nA1, t2, nA2, t3, nA3, t4, nA4)
ILCCTDEMON5(name, IlcCtClass, IlcFnName, t1, nA1, t2, nA2, t3, nA3, t4, nA4, t5,
nA5)
ILCCTDEMON6(name, IlcCtClass, IlcFnName, t1, nA1, t2, nA2, t3, nA3, t4, nA4, t5,
nA5, t6, nA6)
```

This macro defines a demon class named `nameI` with *n* data members. When *n* is greater than 0, the types and names of the data members must be supplied as arguments to the macro. Each data member is defined by its type `T`*i* and a name `data`*i*. Besides the definition of the class `nameI`, this macro also defines a function named `name` that creates an instance of the class `nameI` and that returns an instance of the class `IlcDemon` that points to it.

An instance of a class of demons created in this way can serve as a *parent* of constraints. The member function `IlcConstraint::getParentDemon` accesses the demon-parent of a constraint.

You are not obliged to use this macro to define demons. When the macro seems too restrictive for your purposes, we recommend that you define a demon class directly.

Since the argument `name` is used to name the demon class, it is not possible to use the same name for several demon definitions.

Example

Here's how to define a demon that calls the function `MyConstraintI::reduceDomain(IlcIntVar var);` of the constraint `ct`:

```
ILCCTDEMON1 (CallReduceDomain, MyConstraintI,
          reduceDomain, IlcIntVar, var);
```

This macro then generates code similar to the following lines:

```
class CallReduceDomainI: public IlcDemonI {
     IlcIntVar var;
public:
     CallReduceDomainI(IloSolver s,
                       MyConstraintI* ct,
                       IlcIntVar avar):
     IlcDemonI(s,ct), var(avar) {}
     ~CallReduceRomainI(){}
     void propagate();
};
IlcDemon CallReduceDomain(IloSolver s,
                       MyConstraintI* ct,
                       IlcIntVar var){
     return new (s.getHeap())
                 CallReduceDomainI(s,ct,var);
}
void CallReduceDomainI::propagate(){
     ((MyConstraintI*)getConstraint())->reduceDomain(var);
}
```

The following statement creates an instance of the class `CallReduceDomainI` and returns a handle that points to it.

```
CallReduceDomain(s,ct,var);
```

For more information, see the concepts Propagation and Propagation Events.

See Also: IlcConstraintI, IlcDemonI, IlcGoalI

Macro ILCGOAL0

Definition file: ilsolver/basic.h

ILCGOAL0(name)
ILCGOAL1(name, t1, nA1)
ILCGOAL2(name, t1, nA1, t2, nA2)
ILCGOAL3(name, t1, nA1, t2, nA2, t3, nA3)
ILCGOAL4(name, t1, nA1, t2, nA2, t3, nA3, t4, nA4)
ILCGOAL5(name, t1, nA1, t2, nA2, t3, nA3, t4, nA4, t5, nA5)
ILCGOAL6(name, t1, nA1, t2, nA2, t3, nA3, t4, nA4, t5, nA5, t6, nA6)

This macro defines a goal class named `nameI` with *n* data members. When *n* is greater than 0 (zero), the types and names of the data members must be supplied as arguments to the macro. Each data member is defined by its type `T`*i* and a name `data`*i*. The call to the macro must be followed immediately by the body of the `execute` member function of the goal class being defined. Besides the definition of the class `nameI`, this macro also defines a function named `name` whose first argument is `IloSolver s`, and the *n* following ones correspond to the *n* data members. This function creates an instance of the class `nameI`, fills the data members with its *n* last arguments, and returns an instance of the class `IlcGoal` that points to it.

You are not obliged to use this macro to define goals. When the macro seems too restrictive for your purposes, we recommend that you define a goal class directly.

Since the argument `name` is used to name the goal class, it is not possible to use the same name for several goal definitions.

Examples:

Here's how to define a goal with one data member:

```
ILCGOAL1(PrintX, IlcInt, x){
  IloSolver s = getSolver();
  s.out() << "PrintX: a goal with one data member" << endl;
  s.out() << x << endl;
  return 0;
}
```

This macro generates code similar to the following lines:

```
class PrintXI : public IlcGoalI {
  public:
    IlcInt x;
    PrintXI(IloSolver solver, IlcInt);
    IlcGoal execute();
};
PrintXI::PrintXI(IloSolver solver, IlcInt arg1)
  :IlcGoalI(solver), x(arg1){}
IlcGoal PrintX(IloSolver s, IlcInt x){
  return new (s.getHeap()) PrintXI(s.getImpl(), x);
}
IlcGoal PrintXI :: execute() {
  IloSolver s = getSolver();
  s.out() << "PrintX: a goal with one data member" << endl;
  s.out() << x << endl;
  return 0;
}
```

The following statement creates an instance of the class `PrintXI` and returns a handle that points to it.

```
PrintX(s, 2);
```

See Also: IlcGoal, IlcGoalI

Macro IlcHalfPi

Definition file: ilsolver/linfloat.h

`IlcHalfPi`

This global floating-point constant is an approximation of one-half of Pi.

See Also: IlcArcCos, IlcArcSin, IlcArcTan, IlcCos, IlcPi, IlcQuarterPi, IlcSin, IlcTan, IlcThreeHalfPi, IlcTwoPi

Macro IlcInfinity

Definition file: ilsolver/ilcerr.h

`IlcInfinity`

This global floating-point constant is equal to the IEEE 754 special value “plus infinity.” Depending on the compiler, this constant is displayed variously as `Infinity, ++, inf`, etc.

IEEE 754 is a standard proposed by the Institute of Electronic and Electrical Engineers for computing floating-point arithmetic. The implementation of floating-point numbers in Solver conforms to this standard. See the Solver User's Manual for a discussion of floating-point arithmetic.

See Also: IlcFloat, IlcFloatMax

Macro IlcIntMax

Definition file: ilsolver/ilcerr.h

`IlcIntMax`

This constant represents the largest possible positive integer on a given platform.

The member function `IlcIntExp::getSize` (indicating the number of values in the domain of an integer expression) returns `IlcIntMax` whenever `max - min` (the difference between the upper and lower bounds of the domain of the expression) is greater than `IlcIntMax`.

If an integer expression overflows positively (that is, if a bound is greater than `IlcIntMax`) then Solver replaces that bound by `IlcIntMax`.

Solver evaluates the expression `0/0` as the interval `[IlcIntMin..IlcIntMax]`.

See Also: IlcInt, IlcIntMin

Macro IlcIntMin

Definition file: ilsolver/ilcerr.h

`IlcIntMin`

This constant represents the smallest possible negative integer on a given platform.

If an integer expression (that is, an instance of `IlcIntExp` or one of its subclasses) overflows negatively (that is, if a bound is less than `IlcIntMin`), then Solver replaces that bound by `IlcIntMin`.

The value `IlcIntMin - 1` (sometimes known as the Joker) is treated correctly.

Solver evaluates the expression `0/0` as the interval `[IlcIntMin..IlcIntMax]`.

See Also: IlcInt, IlcIntMax

Macro ILCINTPREDICATE0

Definition file: ilsolver/ilcint.h

ILCINTPREDICATE0(name)
ILCINTPREDICATE1(name, type1, nameArg1)
ILCINTPREDICATE2(name, type1, nameArg1, type2, nameArg2)
ILCINTPREDICATE3(name, type1, nameArg1, type2, nameArg2, type3, nameArg3)
ILCINTPREDICATE4(name, type1, nameArg1, type2, nameArg2, type3, nameArg3, type4, nameArg4)
ILCINTPREDICATE5(name, type1, nameArg1, type2, nameArg2, type3, nameArg3, type4, nameArg4, type5, nameArg5)
ILCINTPREDICATE6(name, type1, nameArg1, type2, nameArg2, type3, nameArg3, type4, nameArg4, type5, nameArg5, type6, nameArg6)

This macro defines an integer predicate class named `nameI` with *n* data members. When *n* is greater than 0 (zero), the types and names of the data members must be supplied as arguments to the macro. Each data member is defined by its type `T`*i* and a name `data`*i*. The call to the macro must be followed immediately by the body of the `isTrue` member function of the integer predicate class being defined. Besides the definition of the class `nameI`, this macro also defines a function named `name` that creates an instance of the class `nameI` and that returns an instance of the class `IlcIntPredicate` that points to it.

Solver does not check the arity of the predicate that you defined. It assumes that the size of the array (an instance of `IlcIntArray`) passed as an argument to the member function `IlcIntPredicate::isTrue` will always be the same. It also assumes that the name of the array passed as an argument is `val`. That is, you *must* use that name to define a predicate.

You are not obliged to use this macro to define integer predicates. When the macro seems too restrictive for your purposes, we recommend that you define an integer predicate class directly.

Since the argument `name` is used to name the integer predicate class, it is not possible to use the same name for several integer predicate definitions.

Example

Here's how to define an integer predicate with one data member:

```
ILCINTPREDICATE1(AllLessThanX, IlcInt, x){
    return (val[0] < x && val[1] < x && val[2] <
x);
}
```

That predicate is a ternary predicate, so it assumes that the array passed an argument to the member function `IlcIntPredicate::isTrue` is of size three. The predicate is true if all the values are less than the integer x.

That macro generates code similar to the following lines:

```
class AllLessThanXI : public IlcIntPredicateI {
    IlcInt x;
public:
    AllLessThanXI(IlcInt xx):x(xx){}
    ~AllLessThanXI(){}
    IlcBool isTrue(IlcIntArray val);
};
IlcIntPredicate AllLessThanX(IloSolver s, IlcInt xx){
    return new (s.getHeap()) AllLessThanXI(xx);
}
IlcBool AllLessThanXI::isTrue(IlcIntArray val){
    return (val[0] < x && val[1] < x && val[2] <
x);
}
```

The following statement creates an instance of the class `AllLessThanXI` and returns a handle that points to it.

```
AllLessThanX(s, 4);
```

See Also: IlcIntArray, IlcIntPredicate, IlcIntPredicateI, IlcTableConstraint

Macro IlcPi

Definition file: ilsolver/linfloat.h

`IlcPi`

This global floating-point constant is an approximation of Pi.

See Also: IlcArcCos, IlcArcSin, IlcArcTan, IlcCos, IlcDegToRad, IlcHalfPi, IlcQuarterPi, IlcRadToDeg, IlcSin, IlcTan, IlcThreeHalfPi, IlcTwoPi

Macro IlcQuarterPi

Definition file: ilsolver/linfloat.h

`IlcQuarterPi`

This global floating-point constant approximates one-quarter of Pi.

See Also: IlcArcCos, IlcArcSin, IlcArcTan, IlcCos, IlcDegToRad, IlcHalfPi, IlcPi, IlcRadToDeg, IlcSin, IlcTan, IlcThreeHalfPi, IlcTwoPi

Macro ILCREV

Definition file: ilsolver/basic.h

ILCREV(t)

This macro defines a reversible class for pointers to objects of the class indicated by its argument `type`. The macro is currently useful because the class `IlcRevAny` is not typed; the pointers in objects of that class are of type `IlcInt` and thus give no indication of the type of objects they point to. In future releases, this macro will be replaced by a class template. Solver does not yet use templates because they are not yet correctly supported by all C++ compilers.

This macro makes use of `IlcRev`, the root class of the reversible classes `IlcRevAny`, `IlcRevBool`, `IlcRevFloat`, and `IlcRevInt`. That is, `IlcRev` is the base class in the macro. This class is not intended for direct instantiation. In other words, you should not call its constructors directly yourself. Here is its synopsis.

```
class IlcRev{
  public:
    IlcRev();
    IlcRev(IloSolver s);
    ~IlcRev(){}
};
```

To see how the macro works, let's assume that `T` is the name of a class. Then the following statement creates a class of objects that each behave as a reversible pointer to an object of class `T`.

```
ILCREV(T);
```

That statement defines the following class, `IlcRevT`.

```
class IlcRevT : public IlcRev{
  public:
    operator T * () const;
    T * operator ->() const;
    T * getValue() const;
    void setValue(IloSolver solver, T * value);
};
IlcBool operator == (const IlcRevT& rev, T * cst);
IlcBool operator == (T * cst, const IlcRevT& rev);
IlcBool operator == (const IlcRevT& rev1, const IlcRevT& rev2);
IlcBool operator != (const IlcRevT& rev, T * cst);
IlcBool operator != (T * cst, const IlcRevT& rev);
IlcBool operator != (const IlcRevT& rev1, const IlcRevT&
rev2);
```

The operators and member functions of that class behave like those of the class `IlcRevInt`, with the exception of the operator `->`.

```
T* IlcRevT::operator ->() const;
```

This overloaded operator returns the address of the object pointed to by the invoking object of `IlcRevT`.

Example

Here's an example showing how to have a data member for which all modifications are reversible.

```
class Dummy;

ILCREV(Dummy);

class Dummy {
  IloSolver _s;
  IlcRevInt _data;   // reversible
  IlcRevDummy _next; // reversible
  IlcInt _state;     // not reversible
public:
  Dummy(IloSolver s, IlcInt i, IlcInt state):
```

```
    _s(s),_data(i), _state(state){}
  ~Dummy(){}
  IlcInt getData() const { return _data; }
  void setData(IlcInt i) {
    _data.setValue(_s, i);
  }
  Dummy* getNext() const { return _next; }
  void setNext(Dummy* next) {
    _next.setValue(_s, next);
  }
  IlcInt getNextData() const;
};
IlcInt Dummy::getNextData() const {
  if (_next)
    return _next->getData(); // uses overloaded ->
}
```

The value of data members of that class will be restored automatically by Solver when it backtracks because they have been defined with the reversible classes, `IlcRevInt` and `IlcRevDummy`.

For more information, see the concepts State and Reversibility.

See Also: IlcRevAny

Macro ILCSTLBEGIN

Definition file: ilsolver/ilcerr.h

`ILCSTLBEGIN`

This macro enables you run examples either with the STL (Standard Template Library) of Microsoft Visual C++ or with other platforms. ILCSTLBEGIN is present after the last `#include` of each source example. It is defined as:

```
using namespace std
```

when the STL is used (ports of type `stat_sta`, `stat_mta` or `stat_mda`); otherwise, its value is simply null.

Macro IlcThreeHalfPi

Definition file: ilsolver/linfloat.h

`IlcThreeHalfPi`

This global floating-point constant approximates one and a half times the value of Pi.

See Also: IlcArcCos, IlcArcSin, IlcArcTan, IlcCos, IlcDegToRad, IlcHalfPi, IlcQuarterPi, IlcPi, IlcRadToDeg, IlcSin, IlcTan, IlcTwoPi

Macro IlcTwoPi

Definition file: ilsolver/linfloat.h

`IlcTwoPi`

This global floating-point constant approximates twice the value of Pi.

See Also: IlcArcCos, IlcArcSin, IlcArcTan, IlcCos, IlcDegToRad, IlcHalfPi, IlcQuarterPi, IlcPi, IlcRadToDeg, IlcSin, IlcTan, IlcThreeHalfPi

Macro ILOANYBINARYPREDICATE0

Definition file: ilconcert/ilotupleset.h

ILOANYBINARYPREDICATE0(name)
ILOANYBINARYPREDICATE1(name, type1, nameArg1)
ILOANYBINARYPREDICATE2(name, type1, nameArg1, type2, nameArg2)
ILOANYBINARYPREDICATE3(name, type1, nameArg1, type2, nameArg2, type3, nameArg3)
ILOANYBINARYPREDICATE4(name, type1, nameArg1, type2, nameArg2, type3, nameArg3, type4, nameArg4)
ILOANYBINARYPREDICATE5(name, type1, nameArg1, type2, nameArg2, type3, nameArg3, type4, nameArg4, type5, nameArg5)
ILOANYBINARYPREDICATE6(name, type1, nameArg1, type2, nameArg2, type3, nameArg3, type4, nameArg4, type5, nameArg5, type6, nameArg6)

For IBM ILOG Solver: defines a binary predicate class.
This macro defines a predicate class named `nameI` with *n* data members for use in a model. When *n* is greater than 0 (zero), the types and names of the data members must be supplied as arguments to the macro. Each data member is defined by its type `T`*i* and a name `data`*i*. The call to the macro must be followed immediately by the body of the `isTrue` member function of the predicate class being defined. Besides the definition of the class `nameI`, this macro also defines a function named `name` that creates an instance of the class `nameI` and that returns an instance of the class `IloAnyBinaryPredicate` that points to it.

You are not obliged to use this macro to define binary predicates on arbitrary objects. When the macro seems too restrictive for your purposes, we recommend that you define a predicate class directly by subclassing `IlcAnyPredicateI`, documented in the *IBM ILOG Solver Reference Manual*.

Since the argument `name` is used to name the predicate class, it is not possible to use the same name for several predicate definitions.

See Also: IloAnyBinaryPredicate

Macro ILOANYTERNARYPREDICATE0

Definition file: ilconcert/ilotupleset.h

```
ILOANYTERNARYPREDICATE0(name)
ILOANYTERNARYPREDICATE1(name, type1, nameArg1)
ILOANYTERNARYPREDICATE2(name, type1, nameArg1, type2, nameArg2)
ILOANYTERNARYPREDICATE3(name, type1, nameArg1, type2, nameArg2, type3, nameArg3)
ILOANYTERNARYPREDICATE4(name, type1, nameArg1, type2, nameArg2, type3, nameArg3,
type4, nameArg4)
ILOANYTERNARYPREDICATE5(name, type1, nameArg1, type2, nameArg2, type3, nameArg3,
type4, nameArg4, type5, nameArg5)
ILOANYTERNARYPREDICATE6(name, type1, nameArg1, type2, nameArg2, type3, nameArg3,
type4, nameArg4, type5, nameArg5, type6, nameArg6)
```

For IBM® ILOG® Solver: defines a ternary predicate class.
This macro defines a predicate class named `nameI` with *n* data members for use in a model. When *n* is greater than 0, the types and names of the data members must be supplied as arguments to the macro. Each data member is defined by its type `T`*i* and a name `data`*i*. The call to the macro must be followed immediately by the body of the `isTrue` member function of the predicate class being defined. Besides the definition of the class `nameI`, this macro also defines a function named `name` that creates an instance of the class `nameI` and that returns an instance of the class `IloAnyTernaryPredicate` that points to it.

You are not obliged to use this macro to define ternary predicates on arbitrary objects. When the macro seems too restrictive for your purposes, we recommend that you define a predicate class directly by subclassing `IlcAnyPredicateI` (documented in the *IBM ILOG Solver Reference Manual*).

Since the argument `name` is used to name the predicate class, it is not possible to use the same name for several predicate definitions.

See Also: IloAnyTernaryPredicate

Macro IloChooseIntIndex

Definition file: ilsolver/ilosolverint.h

`IloChooseIntIndex`

This macro creates a pointer to a function that takes an array of constrained integer expressions and returns an integer.

In its search primitives for finding solutions, Solver lets you set parameters for choosing the order in which constrained variables are bound. You do so by means of choice functions called *criteria*.

See Also: IloBestGenerate, IloChooseFirstUnboundInt, IlcChooseIntIndex, IlcChooseMaxMaxInt, IlcChooseMaxMinInt, IlcChooseMaxRegretMax, IlcChooseMaxRegretMin, IlcChooseMaxSizeInt, IlcChooseMinMaxInt, IlcChooseMinMinInt, IlcChooseMinRegretMax, IlcChooseMinRegretMin, IlcChooseMinSizeInt, IloGenerate

Macro ILOCLIKECOMPARATOR0

Definition file: ilsolver/iloselector.h

ILOCLIKECOMPARATOR0`(comparatorName, tx, nx1, nx2)`
ILOCLIKECOMPARATOR1`(comparatorName, tx, nx1, nx2, td, nd)`
ILOCLIKECOMPARATOR2`(comparatorName, tx, nx1, nx2, td1, nd1, td2, nd2)`
ILOCLIKECOMPARATOR3`(comparatorName, tx, nx1, nx2, td1, nd1, td2, nd2, td3, nd3)`
ILOCLIKECOMPARATOR4`(comparatorName, tx, nx1, nx2, td1, nd1, td2, nd2, td3, nd3, td4, nd4)`
ILOCLIKECOMPARATOR5`(comparatorName, tx, nx1, nx2, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5)`
ILOCLIKECOMPARATOR6`(comparatorName, tx, nx1, nx2, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5, td6, nd6)`
ILOCLIKECOMPARATOR7`(comparatorName, tx, nx1, nx2, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5, td6, nd6, td7, nd7)`
ILOCTXCLIKECOMPARATOR0`(comparatorName, tx, nx1, nx2, tu, nu)`
ILOCTXCLIKECOMPARATOR1`(comparatorName, tx, nx1, nx2, tu, nu, td, nd)`
ILOCTXCLIKECOMPARATOR2`(comparatorName, tx, nx1, nx2, tu, nu, td1, nd1, td2, nd2)`
ILOCTXCLIKECOMPARATOR3`(comparatorName, tx, nx1, nx2, tu, nu, td1, nd1, td2, nd2, td3, nd3)`
ILOCTXCLIKECOMPARATOR4`(comparatorName, tx, nx1, nx2, tu, nu, td1, nd1, td2, nd2, td3, nd3, td4, nd4)`
ILOCTXCLIKECOMPARATOR5`(comparatorName, tx, nx1, nx2, tu, nu, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5)`
ILOCTXCLIKECOMPARATOR6`(comparatorName, tx, nx1, nx2, tu, nu, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5, td6, nd6)`
ILOCTXCLIKECOMPARATOR7`(comparatorName, tx, nx1, nx2, tu, nu, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5, td6, nd6, td7, nd7)`

The `ILOCLIKECOMPARATORi` macros can be used to generate comparators over arbitrary objects.

The macros take four mandatory parameters:

- The name of the comparator to generate
- The type of the objects to compare
- The name of the first object to compare
- The name of the second object to compare

After this, depending on the macro used, optional parameters can be provided as argument pairs of *type* and *name*.

The body of the macro must return an integer which is equal to `-1` if the comparator's left-hand side is better than its right-hand side, to `1` if the comparator's left-hand side is worse than its right-hand side, and to `0` otherwise.

The macro `ILOCTXCLIKECOMPARATORi` can be used to handle a user-given context (name `nu` and type `tu`) at comparison time.

See macro `ILOCOMPARATOR0` for an example of comparators defined using a macro.

For more information, see Selectors.

Macro ILOCOMPARATOR0

Definition file: ilsolver/iloselector.h

ILOCOMPARATOR0(comparatorName, tx, nx1, nx2)
ILOCOMPARATOR1(comparatorName, tx, nx1, nx2, td, nd)
ILOCOMPARATOR2(comparatorName, tx, nx1, nx2, td1, nd1, td2, nd2)
ILOCOMPARATOR3(comparatorName, tx, nx1, nx2, td1, nd1, td2, nd2, td3, nd3)
ILOCOMPARATOR4(comparatorName, tx, nx1, nx2, td1, nd1, td2, nd2, td3, nd3, td4, nd4)
ILOCOMPARATOR5(comparatorName, tx, nx1, nx2, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5)
ILOCOMPARATOR6(comparatorName, tx, nx1, nx2, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5, td6, nd6)
ILOCOMPARATOR7(comparatorName, tx, nx1, nx2, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5, td6, nd6, td7, nd7)
ILOCTXCOMPARATOR0(comparatorName, tx, nx1, nx2, tu, nu)
ILOCTXCOMPARATOR1(comparatorName, tx, nx1, nx2, tu, nu, td, nd)
ILOCTXCOMPARATOR2(comparatorName, tx, nx1, nx2, tu, nu, td1, nd1, td2, nd2)
ILOCTXCOMPARATOR3(comparatorName, tx, nx1, nx2, tu, nu, td1, nd1, td2, nd2, td3, nd3)
ILOCTXCOMPARATOR4(comparatorName, tx, nx1, nx2, tu, nu, td1, nd1, td2, nd2, td3, nd3, td4, nd4)
ILOCTXCOMPARATOR5(comparatorName, tx, nx1, nx2, tu, nu, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5)
ILOCTXCOMPARATOR6(comparatorName, tx, nx1, nx2, tu, nu, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5, td6, nd6)
ILOCTXCOMPARATOR7(comparatorName, tx, nx1, nx2, tu, nu, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5, td6, nd6, td7, nd7)

The `ILOCOMPARATORi` macros can be used to generate comparators over arbitrary objects.

The macros take four mandatory parameters which are:

- The name of the comparator to generate
- The type of the objects to compare
- The name of the first object to compare
- The name of the second object to compare

After this, depending on the macro used, optional parameters can be provided as argument pairs of *type* and *name*.

The body of the macro must return a Boolean value (`IloBool`) which is `IloTrue` if the comparator's left-hand side is better than its right-hand side.

Using ILOCOMPARATORi

For example, the following code defines a comparator that compares the value of two `IlcFloatVar` variables (provided the variables are bound):

```
ILOCOMPARATOR0(CompareGreaterThan, IlcFloatVar, v1, v2) {
  return v1.getValue() > v2.getValue();
}
```

This comparator can be invoked as follows:

```
IloComparator<IlcFloatVar> cmp = CompareGreaterThan(getSolver());
IloBool value = cmp(var1, var2);
```

Using ILOCTXCOMPARATORi

The macro `ILOCTXCOMPARATORi` can be used to handle a user-given context (name `nu` and type `tu`) at comparison time. The following comparator uses a contextual tolerance factor:

```
ILOCTXCOMPARATOR0(CompareGreaterThanWithTolerance,IlcFloatVar, v1, v2, IlcFloat, tolerance) {
  return (v1.getValue() - v2.getValue() > tolerance);
}
```

This comparator can be invoked as follows:

```
IloComparator<IlcFloatVar> cmp = CompareGreaterThanWithTolerance(getSolver());
IlcFloat tolerance = 0.8;
IloBool value = cmp(var1, var2, (IloAny)&tolerance);
```

For more information, see Selectors.

Macro ILOCPCONSTRAINTWRAPPER0

Definition file: ilsolver/ilosolverint.h

ILOCPCONSTRAINTWRAPPER0`(_this, solver)`
ILOCPCONSTRAINTWRAPPER1`(_this, solver, t1, a1)`
ILOCPCONSTRAINTWRAPPER2`(_this, solver, t1, a1, t2, a2)`
ILOCPCONSTRAINTWRAPPER3`(_this, solver, t1, a1, t2, a2, t3, a3)`
ILOCPCONSTRAINTWRAPPER4`(_this, solver, t1, a1, t2, a2, t3, a3, t4, a4)`
ILOCPCONSTRAINTWRAPPER5`(_this, solver, t1, a1, t2, a2, t3, a3, t4, a4, t5, a5)`
ILOCPCONSTRAINTWRAPPER6`(_this, solver, t1, a1, t2, a2, t3, a3, t4, a4, t5, a5, t6, a6)`

This macro defines a constraint class named `_thisI` with *n* data members. When *n* is greater than zero, the types and names of the data members must be supplied as arguments to the macro. Each data member is defined by its type `t`*i* and a name `a`*i*.

Since the argument `this` is used to name the constraint class, it is not possible to use the same name for several constraints.

You can use the macro `ILOCPCONSTRAINTWRAPPER` to wrap an existing instance of `IlcConstraint` when you want to use it within Concert Technology model objects. In order to use an instance of `IlcConstraint` in that way, you need to follow these steps:

1. Use the macro to wrap the instance of `IlcConstraint` in an instance of `IloConstraint`.
2. Extract your model and model objects for an instance of `IloSolver` by calling the member function `IloSolver::extract`. During extraction, `IloSolver::extract` will put back the instance of `IloConstraint`.

You must use the following IloCPConstraintI member functions to force extraction of an extractable or an array of extractables:

```
void use(const IloSolver solver, const IloExtractable ext)const;
void use(const IloSolver solver, const IloExtractableArray
extArray)const;
```

Example

Here's how to define a constraint wrapper with one data member:

```
ILOCPCONSTRAINTWRAPPER1(IloFreqConstraint, solver, IloIntVarArray, _vars) {
  use(solver, _vars);
  return IlcFreqConstraint(solver, solver.getIntVarArray(_vars));
}
```

In order to use `IloSolver::getIntVarArray`, the object must be extracted. To force extraction of an extractable or an array of extractables, pass them as parameters of the `use` method.

That macro generates code similar to the following lines:

```
class IloFreqConstraintI : public IloCPConstraintI {

  ILOCPCONSTRAINTWRAPPERDECL

private:

  IloIntVarArray _vars;

public:

  IloFreqConstraintI(IloEnvI*, const IloIntVarArray&, const char*);

  virtual IloExtractableI* makeClone(IloEnvI*) const;
```

```
  virtual void display(ostream       & out) const;

  IlcConstraint extract(const IloSolver) const;

};

ILOCPCONSTRAINTWRAPPERIMPL(IloFreqConstraintI)

IloFreqConstraintI::IloFreqConstraintI(IloEnvI* env,

                                    const IloIntVarArray&   T_vars,

                                    const char* name) :

  IloCPConstraintI (env, name), _vars ((IloIntVarArray&)T_vars) {}

IloExtractableI* IloFreqConstraintI::makeClone(IloEnvI* env) const {

  IloIntVarArray targ1 = IloGetClone(env, _vars);

  return new (env) IloFreqConstraintI(env,

                                   (const IloIntVarArray &)targ1,

                                   (const char*)0);

}

void IloFreqConstraintI::display(ostream& out) const {

  out << "IloFreqConstraintI" << " (";

  if (getName()) out << getName();

  else out << getId(); out << ")" << endl;

  out << "  " << "_vars" << " " << _vars <<
endl;

}

IloConstraint  IloFreqConstraint(IloEnv env,

                              IloIntVarArray _vars,

                              const char* name=0) {

  IloFreqConstraintI::InitTypeIndex();

  return new (env) IloFreqConstraintI(env.getImpl(), _vars, name);

}

IlcConstraint IloFreqConstraintI::extract(const IloSolver& solver) const
{

  use(solver, _vars);

  return IlcFreqConstraint(solver, solver.getIntVarArray(_vars));

}
```

See Also: IloSolver, IloCPConstraintI

Macro ILOCPGOALWRAPPER0

Definition file: ilsolver/ilosolverint.h

ILOCPGOALWRAPPER0(_this, solver)
ILOCPGOALWRAPPER1(_this, solver, t1, a1)
ILOCPGOALWRAPPER2(_this, solver, t1, a1, t2, a2)
ILOCPGOALWRAPPER3(_this, solver, t1, a1, t2, a2, t3, a3)
ILOCPGOALWRAPPER4(_this, solver, t1, a1, t2, a2, t3, a3, t4, a4)
ILOCPGOALWRAPPER5(_this, solver, t1, a1, t2, a2, t3, a3, t4, a4, t5, a5)

This macro defines a goal class named `_thisI` with *n* data members. When *n* is greater than zero, the types and names of the data members must be supplied as arguments to the macro. Each data member is defined by its type `t`*i* and a name `a`*i*.

You can use the macro `ILOCPGOALWRAPPER` to wrap an existing instance of `IlcGoal` when you want to use it within Concert Technology model objects. In order to use an instance of `IlcGoal` in that way, you need to follow these steps:

1. Use the macro to wrap the instance of `IlcGoal` in an instance of `IloGoal`.
2. Use the `IloGoal` generated by `IloSolver::solve` and `IloSolver:startNewSearch`.

Example

Here's how to define a goal wrapper with one data member:

```
ILOCPGOALWRAPPER1(MyGenerate, solver, IloIntVarArray, vars) {
  return IlcGenerate(solver.getIntVarArray(vars), IlcChooseMinSizeMin);
}
```

That macro generates code similar to the following lines:

```
class MyGenerateConcertI : public IloGoalI {
  IloIntVarArray vars ;
public:
  MyGenerateConcertI (IloEnvI*, IloIntVarArray );
  ~MyGenerateConcertI ();
  virtual IlcGoal extract(const IloSolver) const;
  virtual IloGoalI* makeClone(IloEnvI*) const;
};

MyGenerateConcertI::MyGenerateConcertI(IloEnvI* env,
                                    IloIntVarArray varsvars) :
  IloGoalI(env), vars (varsvars) {}

MyGenerateConcertI::~MyGenerateConcertI () {}

IloGoalI* MyGenerateConcertI::makeClone(IloEnvI* env) const {
  IloIntVarArray na1 = IloGetClone(env, vars);
  return new (env) MyGenerateConcertI(env, na1);
}

IloGoal MyGenerate (IloEnv env, IloIntVarArray varsvars) {
  return new (env) MyGenerateConcertI (env.getImpl(), varsvars);
}

IlcGoal MyGenerateConcertI::extract(const IloSolver solver) const {
  return IlcGenerate(solver.getIntVarArray(vars), IlcChooseMinSizeMin);
}
```

The extraction of a goal does not extract and must not extract its arguments. Therefore, when using a goal you must make sure that the arguments will be extracted by the model. A good way to ensure this is to add the arguments to the models.

This is illustrated by the following example:

```
IloEnv env;
```

```
IloModel model(env);
IloIntVar x(env, 0, 10);
model.add(x); // This is mandatory otherwise the variable will not be
extracted
IloSolver solver(model);
solver.solve(IloInstantiate(env, x));
env.end();
```

See Also: IloSolver

Macro ILOCPTRACEWRAPPER0

Definition file: ilsolver/ilosolverint.h

ILOCPTRACEWRAPPER0(_this, solver)
ILOCPTRACEWRAPPER1(_this, solver, t1, a1)
ILOCPTRACEWRAPPER2(_this, solver, t1, a1, t2, a2)
ILOCPTRACEWRAPPER3(_this, solver, t1, a1, t2, a2, t3, a3)
ILOCPTRACEWRAPPER4(_this, solver, t1, a1, t2, a2, t3, a3, t4, a4)
ILOCPTRACEWRAPPER5(_this, solver, t1, a1, t2, a2, t3, a3, t4, a4, t5, a5)
ILOCPTRACEWRAPPER6(_this, solver, t1, a1, t2, a2, t3, a3, t4, a4, t5, a5, t6, a6)

This macro defines a trace class named `_thisI` with *n* data members. When *n* is greater than zero, the types and names of the data members must be supplied as arguments to the macro. Each data member is defined by its type `t`*i* and a name `a`*i*.

You can use the macro `ILOCPTRACEWRAPPER` to wrap an existing instance of `IlcTrace` when you want to use it within Concert Technology objects. In order to use an instance of `IlcTrace` in that way, you need to follow these steps:

1. Use the macro to wrap the instance of `IlcTrace` in an instance of `IloCPTrace`.
2. Use `IloSolver::addTrace` to add the trace to the model.

Example

Here is how to define a trace wrapper with no (zero) data members:

```
ILOCPTRACEWRAPPER0(PrintConstraintTrace, solver) {
  solver.setTraceMode(IlcTrue);
  IlcPrintTrace trace(solver, IlcTraceConstraint);
  solver.setTrace(trace);
}
```

That macro generates code similar to the following lines:

```
class PrintConstraintTraceConcertI : public IloCPTraceI {
public:
  PrintConstraintTraceConcertI();
  ~PrintConstraintTraceConcertI();
  virtual void execute(const IloSolver solver) const;
};
PrintConstraintTraceConcertI::PrintConstraintTraceConcertI() :
  IloCPTraceI() {}
PrintConstraintTraceConcertI::~PrintConstraintTraceConcertI() {}
IloCPTrace PrintConstraintTrace(IloEnv env) {
  return new (env) PrintConstraintTraceConcertI();
}
void PrintConstraintTraceConcertI::execute(const IloSolver solver) const
{
  solver.setTraceMode(IlcTrue);
  IlcPrintTrace trace(solver, IlcTraceConstraint);
  solver.setTrace(trace);
}
```

See Also: IloCPTrace, IloCPTraceI

Macro ILODECLDEFAULTCOMPARATOR

Definition file: ilsolver/iimmulti.h

ILODECLDEFAULTCOMPARATOR(IloObject)

Macro for declaring a default comparator.
This macro declares a special template function which is called when a default comparator is needed (for instance by class `IloExplicitEvaluator`). It should be used in conjunction with `ILODEFAULTCOMPARATOR` when the defined comparator function is in a different source file. Normally, this macro will be used in a header file.

See Also: ILODEFAULTCOMPARATOR

Macro ILODEFAULTCOMPARATOR

Definition file: ilsolver/iimmulti.h

ILODEFAULTCOMPARATOR`(IloObject, o1, o2)`

Macro for defining a default comparator.
This macro can be used for a *default comparator* for a particular object type. The macro creates a special template function which delivers a comparator which will compare objects of a given type. The macro takes three arguments, the first being the object type and the next two being the names of the two objects to compare. The body of the comparator should be written in exactly the same way as for `ILOCOMPARATOR0`, where a true value is returned if and only if the first parameter is considered smaller than the second. Note that as this comparison is considered to be innate to the object type, no additional parameters can be specified. The generated template function can be called by classes that need to compare objects (normally for equality). For example, see the `IloExplicitEvaluator` class.

See Also: ILODECLDEFAULTCOMPARATOR, IloExplicitEvaluator

Macro ILODEFINELNSFRAGMENT0

Definition file: ilsolver/iimlns.h

ILODEFINELNSFRAGMENT0(NAME, S)
ILODEFINELNSFRAGMENT1(NAME, S, T1, P1)
ILODEFINELNSFRAGMENT2(NAME, S, T1, P1, T2, P2)
ILODEFINELNSFRAGMENT3(NAME, S, T1, P1, T2, P2, T3, P3)
ILODEFINELNSFRAGMENT4(NAME, S, T1, P1, T2, P2, T3, P3, T4, P4)
ILODEFINELNSFRAGMENT5(NAME, S, T1, P1, T2, P2, T3, P3, T4, P4, T5, P5)
ILODEFINELNSFRAGMENT6(NAME, S, T1, P1, T2, P2, T3, P3, T4, P4, T5, P5, T6, P6)
ILODEFINELNSFRAGMENT7(NAME, S, T1, P1, T2, P2, T3, P3, T4, P4, T5, P5, T6, P6, T7, P7)

Macro for more easily creating LNS neighborhoods.
This macro simplifies subclassing the class `IloLargeNHoodI` in order to create large neighborhoods. When you use this macro, you are in fact subclassing `IloLargeNHoodI` and redefining the method `IloLargeNHoodI::defineFragment`. Thus, in the body of the macro, you should make calls to the function `addToFragment` to add each variable you wish to the fragment.

This macro takes two mandatory parameters. The first is the name of a function which will be generated that will create the large neighborhood. The second is the name of an instance of `IloSolver` which is driving the local search. After these two mandatory parameters, depending on the particular version of the macro employed, you pass optional parameters in the usual way using pairs of type and name.

An example of the use of this macro from the `YourSolverHomeexamplessrclstalent.cpp` example is shown below:

```
ILODEFINELNSFRAGMENT1(SegmentLNSNHood, solver, IloIntVarArray, x) {
  IlcRandom r = solver.getRandom();
  IloInt a = r.getInt(x.getSize());
  IloInt b = r.getInt(x.getSize());
  if (a > b) { IloInt tmp = a; a = b; b = tmp; }
  for (IlcInt i = a; i <= b; i++)
    addToFragment(solver, x[i]);
}
```

Macro ILOEVALUATOR0

Definition file: ilsolver/iloselector.h

ILOEVALUATOR0(name, tx, nx)
ILOEVALUATOR1(name, tx, nx, td, nd)
ILOEVALUATOR2(name, tx, nx, td1, nd1, td2, nd2)
ILOEVALUATOR3(name, tx, nx, td1, nd1, td2, nd2, td3, nd3)
ILOEVALUATOR4(name, tx, nx, td1, nd1, td2, nd2, td3, nd3, td4, nd4)
ILOEVALUATOR5(name, tx, nx, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5)
ILOEVALUATOR6(name, tx, nx, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5, td6, nd6)
ILOEVALUATOR7(name, tx, nx, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5, td6, nd6, td7, nd7)
ILOCTXEVALUATOR0(name, tx, nx, tu, nu)
ILOCTXEVALUATOR1(name, tx, nx, tu, nu, td, nd)
ILOCTXEVALUATOR2(name, tx, nx, tu, nu, td1, nd1, td2, nd2)
ILOCTXEVALUATOR3(name, tx, nx, tu, nu, td1, nd1, td2, nd2, td3, nd3)
ILOCTXEVALUATOR4(name, tx, nx, tu, nu, td1, nd1, td2, nd2, td3, nd3, td4, nd4)
ILOCTXEVALUATOR5(name, tx, nx, tu, nu, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5)
ILOCTXEVALUATOR6(name, tx, nx, tu, nu, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5, td6, nd6)
ILOCTXEVALUATOR7(name, tx, nx, tu, nu, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5, td6, nd6, td7, nd7)

The macros `ILOEVALUATORi` and `ILOCTXEVALUATORi` define one function named `name` that creates and returns an instance of an evaluator of objects of class `tx` with *i* data members of type `td1...td`*i*. `ILOCTXEVALUATORi` allows an additional dynamic evaluation context to be passed to be used inside the user-defined evaluation function.

This function's signature returns an evaluator allocated on an environment or on a solver heap:

```
IloEvaluator<tx> name(IloEnv, td1, ..., tdi);
IloEvaluator<tx> name(IloSolver, td1, ..., tdi);
```

Note the important difference between data members and contexts. For a given instance of evaluator, data members represent a unique instance of object linked with the evaluator and given at construction time, whereas the context is given in the evaluation function at evaluation time and thus may change depending on the instance of object that is evaluated:

```
IloEvaluator<tx>::operator()(tx nx, IloAny nu)
```

The two following examples show how to define an evaluator that evaluates the value of an instance of `IloNumVar` in a solution.

Example 1

In the first case, it is assumed that you may have to evaluate variables in different instances of `solution` so that the solution must be given as a context to the evaluation function:

```
ILOCTXEVALUATOR0(ValueInContextualSolution,
                 IloNumVar, v,
                 IloSolution, solution) {
  assert(solution.isBound(v));
  return solution.getValue(v);
}
```

The macro above defines the following function:

```
IloEvaluator<IloNumVar> ValueInContextualSolution(IloEnv);
```

The evaluation function must be given the address of a solution handle as context:

```
IloEnv env;
IloNumVar v = ...;
IloSolution solution = ...;
IloEvaluator<IloNumVar> eval1 = ValueInContextualSolution(env);
IloNum val = eval1(v, &solution);
```

Example 2

In the second case, it is assumed that you only have one instance of `solution` in the problem and that all the evaluations will use this solution so that the solution can be represented as a data member of the evaluator.

```
ILOEVALUATOR1(ValueInTheSolution,
              IloFloatVar, v,
              IloSolution, solution) {
  assert(solution.isBound(v));
  return solution.getValue(v);
}
```

The macro above defines the following function:

```
IloEvaluator<IloNumVar> ValueInTheSolution(IloEnv, IloSolution);
```

The evaluation function does not use any context:

```
IloEnv env;
IloNumVar v = ...;
IloSolution solution = ...;
IloEvaluator<IloNumVar> eval2 = ValueInTheSolution(env, solution);
IloNum val = eval2(v);
```

For more information, see Selectors.

Macro IloFloatArray

Definition file: ilconcert/iloenv.h

IloFloatArray

IloFloatArray is the array class of the basic floating-point class.
IloFloatArray is the array class of the basic floating-point class for a model. It is a handle class. The implementation class for IloFloatArray is the undocumented class IloFloatArrayI.

Instances of IloFloatArray are extensible. (They differ from instances of IlcFloatArray in this respect. IlcFloatArray is documented in the *IBM ILOG Solver Reference Manual* and the *IBM ILOG CP Optimizer Reference Manual*.)

For each basic type, Concert Technology defines a corresponding array class. That array class is a handle class. In other words, an object of that class contains a pointer to another object allocated on the Concert Technology heap associated with a model. Exploiting handles in this way greatly simplifies the programming interface since the handle can then be an automatic object: as a developer using handles, you do not have to worry about memory allocation.

As handles, these objects should be passed by value, and they should be created as automatic objects, where “automatic” has the usual C++ meaning.

Member functions of a handle class correspond to member functions of the same name in the implementation class.

Assert and NDEBUG

Most member functions of the class IloFloatArray are inline functions that contain an assert statement. This statement checks that the handle pointer is not null. These statements can be suppressed by the macro NDEBUG. This option usually reduces execution time. The price you pay for this choice is that attempts to access through null pointers are not trapped and usually result in memory faults.

See Also: IloNum

Macro IloFloatVar

Definition file: ilconcert/iloexpression.h

`IloFloatVar`

An instance of this class represents a constrained floating-point variable in Concert Technology.
An instance of this class represents a constrained floating-point variable in Concert Technology.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloFloatVarArray, IloNumVar

Macro IloFloatVarArray

Definition file: ilconcert/iloexpression.h

`IloFloatVarArray`

The array class of IloFloatVar.
For each basic type, Concert Technology defines a corresponding array class. `IloFloatVarArray` is the array class of the floating-point variable class for a model. It is a handle class.

Instances of `IloFloatVarArray` are extensible.

Most member functions in this class contain `assert` statements. For an explanation of the macro `NDEBUG` (a way to turn on or turn off these `assert` statements), see the concept Assert and NDEBUG.

See Also: IloFloatVar

Macro IloHalfPi

Definition file: ilconcert/ilosys.h

IloHalfPi

Half pi.
Concert Technology predefines conventional trigonometric constants to conform to IEEE 754 standards for quarter pi, half pi, pi, three-halves pi, and two pi.

```
extern const IloNum IloHalfPi;       // = 1.57079632679489661923
```

Macro ILOIIMLISTENER0

Definition file: ilsolver/iimevent.h

ILOIIMLISTENER0(N, ET, E)
ILOIIMLISTENER1(N, ET, E, T1, V1)
ILOIIMLISTENER2(N, ET, E, T1, V1, T2, V2)
ILOIIMLISTENER3(N, ET, E, T1, V1, T2, V2, T3, V3)
ILOIIMLISTENER4(N, ET, E, T1, V1, T2, V2, T3, V3, T4, V4)
ILOIIMLISTENER5(N, ET, E, T1, V1, T2, V2, T3, V3, T4, V4, T5, V5)
ILOIIMLISTENER6(N, ET, E, T1, V1, T2, V2, T3, V3, T4, V4, T5, V5, T6, V6)
ILOIIMLISTENER7(N, ET, E, T1, V1, T2, V2, T3, V3, T4, V4, T5, V5, T6, V6, T7, V7)
ILOIIMLISTENER8(N, ET, E, T1, V1, T2, V2, T3, V3, T4, V4, T5, V5, T6, V6, T7, V7, T8, V8)

A macro to define custom listeners.
The body of this macro defines a piece of code to execute when the object to which the listener is added emits the event in question. No value is returned from a listener.

The listener macro takes three mandatory parameters:

- The name of the listener
- The type of event to listen for
- The name of the event received

Depending on the specific macro used, optional parameters can be added using pairs of arguments specifying the parameter type and the parameter name.

The following code from the example `YourSolverHomeexamplessrcea1max_listen.cpp` defines a listener which monitors operator invocation:

```
ILOIIMLISTENER0(OperatorListener, IloPoolOperator::InvocationEvent, event) {
  IloPoolOperator op = event.getOperator();
  OperatorStatistics * stat = GetOperatorStatistics(op);
  stat->invocations++;
}
```

See Also: IloListener, IloEvent

Macro ILOIIMOP0

Definition file: ilsolver/iimoperator.h

ILOIIMOP0(N, S)

This macro is used to define operators.
This macro is used to define operators. For the most part, operators can be thought of as standard Solver goals with the additional property that they have access to an input pool of solutions which they use to influence their behavior. The macro must be followed by the user defined method body which must return a (possibly null) continuation goal.

When the operator is converted to a processor, the resulting processor will capture the state of the solver as a solution and store it in the output pool of the processor.

A set of `ILOIIMOP0..7` macros allows you to define parameters that can be used in the user defined method. Two parameters are mandatory:

- the name of the operator to be created
- an expression indicating the number of input solutions required by the operator (this can be a numerical constant, or a more general expression)

After the mandatory parameters, optional parameters can be given depending on the version of the macro chosen. These are specified by pairs of object type followed by object name.

Certain services are available to the selector:

- `getSolver()` gets the current solver
- `getInputPool()` delivers the input pool of the operator

The following code from the `YourSolverHomeexamplessrceabinpack.cpp` example begins to define the genetic operator:

```
ILOIIMOP5(PackingOperator, numParents,
          IloInt, numParents,
          IloIntVarArray, load,
          IloIntVarArray, where,
          IloIntArray, weight,
          IloIntVar, used) {
  IloSolver solver = getSolver();
  IloInt numItems = weight.getSize();
  IloInt numBins = solver.getMax(used);
  IloSolutionPool parents = getInputPool();
  IloInt numParents = parents.getSize();
  IloInt reqdMeanLoad = (IloSum(weight) + numBins - 1) / numBins;
  IlcRandom rnd = solver.getRandom();
  IloInt targetBin = rnd.getInt(numBins);
  IloInt currentBin = 0;
  IloInt tries = 0;

  while (currentBin < targetBin && tries < numParents * numBins) {
```

Note that here the second parameter of the macro, which is the number of input solutions the operator requires, is a *variable* which is passed as the first optional parameter of the macro. This is a simple way to write operators which can be instantiated to receive different numbers of parents.

> **Note**
>
> The user defined method will be invoked as many times as necessary to produce the number of solutions required by the `consumer` processor linked to this operator's output.

See Also: IloPoolOperator::operator IloPoolProc, operator>>

Macro ILOINTBINARYPREDICATE0

Definition file: ilconcert/ilotupleset.h

```
ILOINTBINARYPREDICATE0(name)
ILOINTBINARYPREDICATE1(name, type1, nameArg1)
ILOINTBINARYPREDICATE2(name, type1, nameArg1, type2, nameArg2)
ILOINTBINARYPREDICATE3(name, type1, nameArg1, type2, nameArg2, type3, nameArg3)
ILOINTBINARYPREDICATE4(name, type1, nameArg1, type2, nameArg2, type3, nameArg3,
type4, nameArg4)
ILOINTBINARYPREDICATE5(name, type1, nameArg1, type2, nameArg2, type3, nameArg3,
type4, nameArg4, type5, nameArg5)
ILOINTBINARYPREDICATE6(name, type1, nameArg1, type2, nameArg2, type3, nameArg3,
type4, nameArg4, type5, nameArg5, type6, nameArg6)
```

For constraint programming: defines a predicate class.
This macro defines a predicate class named `nameI` with *n* data members for use in a model. When *n* is greater than 0, the types and names of the data members must be supplied as arguments to the macro. Each data member is defined by its type `T`*i* and a name `data`*i*. The call to the macro must be followed immediately by the body of the `isTrue` member function of the predicate class being defined. Besides the definition of the class `nameI`, this macro also defines a function named `name` that creates an instance of the class `nameI` and that returns an instance of the class `IloIntBinaryPredicate` that points to it.

You are not obliged to use this macro to define binary predicates on arbitrary objects. When the macro seems too restrictive for your purposes, we recommend that you define a predicate class directly by subclassing `IlcIntPredicateI`, documented in the *IBM ILOG CP Optimizer Reference Manual* and the *IBM ILOG Solver Reference Manual*.

Since the argument `name` is used to name the predicate class, it is not possible to use the same name for several predicate definitions.

See Also: IloIntBinaryPredicate

Macro ILOINTTERNARYPREDICATE0

Definition file: ilconcert/ilotupleset.h

```
ILOINTTERNARYPREDICATE0(name)
ILOINTTERNARYPREDICATE1(name, type1, nameArg1)
ILOINTTERNARYPREDICATE2(name, type1, nameArg1, type2, nameArg2)
ILOINTTERNARYPREDICATE3(name, type1, nameArg1, type2, nameArg2, type3, nameArg3)
ILOINTTERNARYPREDICATE4(name, type1, nameArg1, type2, nameArg2, type3, nameArg3,
type4, nameArg4)
ILOINTTERNARYPREDICATE5(name, type1, nameArg1, type2, nameArg2, type3, nameArg3,
type4, nameArg4, type5, nameArg5)
ILOINTTERNARYPREDICATE6(name, type1, nameArg1, type2, nameArg2, type3, nameArg3,
type4, nameArg4, type5, nameArg5, type6, nameArg6)
```

For constraint programming: defines a predicate class.
This macro defines a predicate class named `nameI` with *n* data members for use in a model. When *n* is greater than 0, the types and names of the data members must be supplied as arguments to the macro. Each data member is defined by its type `T`*i* and a name `data`*i*. The call to the macro must be followed immediately by the body of the `isTrue` member function of the predicate class being defined. Besides the definition of the class `nameI`, this macro also defines a function named `name` that creates an instance of the class `nameI` and that returns an instance of the class `IloIntTernaryPredicate` that points to it.

You are not obliged to use this macro to define ternary predicates on arbitrary objects. When the macro seems too restrictive for your purposes, we recommend that you define a predicate class directly by subclassing `IlcIntPredicateI` (documented in the *IBM ILOG CP Optimizer Reference Manual* and the *IBM ILOG Solver Reference Manual*).

Since the argument `name` is used to name the predicate class, it is not possible to use the same name for several predicate definitions.

See Also: IloIntTernaryPredicate

Macro ILOMULTIPLEEVALUATOR0

Definition file: ilsolver/iimmulti.h

ILOMULTIPLEEVALUATOR0(name, tx, tc, nc)

Defines an evaluator that performs an evaluation of all objects within a container.
This macro allows you to create an evaluator that performs an evaluation of all objects within a container. The macro defines a function which creates an instance of `IloMultipleEvaluator`, whose purpose is to keep an evaluation of each object in a specified container.

Normally, you would create an instance of `IloEvaluator` to evaluate each object in a container. However, sometimes you need to know *all* the objects at once in order to perform the evaluation, which is when you should use this macro. For example, an evaluator which produces as evaluation the *rank* of an object according to its quality compared to the other solutions requires this type of evaluation.

The four mandatory arguments of this macro are:

- the name of the function to be generated
- the type of object to be evaluated
- the type of container used to contain the objects
- the name of the container

Depending on the exact version of the macro you use, you may then add arguments, which will be present in the generated function, using pairs of `argument-type,argument-name` in the macro.

The function, `setEvaluation` is available to you in the macro, with the same semantics as `IloExplicitEvaluator::setEvaluation`. You should call this function as many times as necessary to fill up the evaluator with evaluations from the pool.

The following code shows a multiple evaluator which takes two additional parameters: another evaluator and a power parameter. The user-defined method raises values obtained from the given evaluator to the given power:

```
// Raise the result of an evaluator to a given power, and normalizes
ILOMULTIPLEEVALUATOR2(PowerEvaluator, IloSolution, IloSolutionPool, pool,
                      IloSolutionPoolEvaluator, eval,
                      IloNum, power) {
  eval.update(pool);
  IloNum sum = 0.0;
  for (IloSolutionPool::Iterator it1(pool); it1.ok(); ++it1)
    sum += pow(eval(*it1), power);
  for (IloSolutionPool::Iterator it2(pool); it2.ok(); ++it2)
    setEvaluation(*it2, pow(eval(*it2), power) / sum);
}
```

Notice that in this code sample, a subordinate multiple evaluator on solutions is passed as parameter. Here, the subordinate evaluator is updated to make sure its values are up to date before using them. This structure is relatively common practice.

See Also: IloEvaluator, IloMultipleEvaluator, IloSolutionPoolEvaluator

Macro IloPi

Definition file: ilconcert/ilosys.h

`IloPi`

Pi.
Concert Technology predefines conventional trigonometric constants to conform to IEEE 754 standards for quarter pi, half pi, pi, three-halves pi, and two pi.

```
extern const IloNum IloPi;           // = 3.14159265358979323846
```

Macro ILOPREDICATE0

Definition file: ilsolver/iloselector.h

ILOPREDICATE0(name, tx, nx)
ILOPREDICATE1(name, tx, nx, td, nd)
ILOPREDICATE2(name, tx, nx, td1, nd1, td2, nd2)
ILOPREDICATE3(name, tx, nx, td1, nd1, td2, nd2, td3, nd3)
ILOPREDICATE4(name, tx, nx, td1, nd1, td2, nd2, td3, nd3, td4, nd4)
ILOPREDICATE5(name, tx, nx, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5)
ILOPREDICATE6(name, tx, nx, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5, td6, nd6)
ILOPREDICATE7(name, tx, nx, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5, td6, nd6, td7, nd7)
ILOCTXPREDICATE0(name, tx, nx, tu, nu)
ILOCTXPREDICATE1(name, tx, nx, tu, nu, td, nd)
ILOCTXPREDICATE2(name, tx, nx, tu, nu, td1, nd1, td2, nd2)
ILOCTXPREDICATE3(name, tx, nx, tu, nu, td1, nd1, td2, nd2, td3, nd3)
ILOCTXPREDICATE4(name, tx, nx, tu, nu, td1, nd1, td2, nd2, td3, nd3, td4, nd4)
ILOCTXPREDICATE5(name, tx, nx, tu, nu, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5)
ILOCTXPREDICATE6(name, tx, nx, tu, nu, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5, td6, nd6)
ILOCTXPREDICATE7(name, tx, nx, tu, nu, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5, td6, nd6, td7, nd7)

The macros `ILOPREDICATEi` and `ILOCTXPREDICATEi` define one function named `name` that creates and returns an instance of a predicate on objects of class `tx` with *i* data members of type `td1...td`*i*. `ILOCTXPREDICATEi` allows an additional context to be passed to be used inside the user-defined test function.

This function's signature returns a predicate allocated on an environment or on a solver heap:

```
IloPredicate<tx> name(IloEnv, td1, ..., tdi)
IloPredicate<tx> name(IloSolver, td1, ..., tdi)
```

Note the important difference between data members and contexts. For a given instance of predicate, data members represent a unique instance of object linked with the predicate and given at construction time, whereas contexts are given in the test function `IloPredicate<tx>::operator()(tx nx, IloAny nu)` and thus may change depending on the instance of object that is tested.

The two examples below show how to define a predicate that tests whether an instance of `IloNumVar` is bound in a solution.

Example 1

In the first case, it is assumed that you may have to test variables in different instances of `solution` so that the solution must be given as a context to the test function:

```
ILOCTXPREDICATE0(IsBoundInContextualSolution,
                 IloNumVar, v,
                 IloSolution, solution) {
  return solution.isBound(v);
}
```

The above macro invocation defines the following function:

```
IloPredicate<IloNumVar> IsBoundInContextualSolution(IloEnv);
```

The test function must be given the address of the solution handle as context:

```
IloEnv env;
IloNumVar v = ...;
```

```
IloSolution solution = ...;
IloPredicate<IloNumVar> pred1 = IsBoundInContextualSolution(env);
IloBool bound = pred1(v, &solution);
```

Example 2

In the second case, it is assumed that you only have one instance of `solution` in the problem and all the tests will use this solution so that the solution can be represented as a data member of the predicate.

```
ILOPREDICATE1(IsBoundInTheSolution,
              IloNumVar, v,
              IloSolution, solution) {
  return solution.isBound(v);
}
```

The macro above defines the following function:

```
IloPredicate<IloNumVar> IsBoundInTheSolution(IloEnv, IloSolution);
```

The test function does not use any context:

```
IloEnv env;
IloNumVar v = ...;
IloSolution solution = ...;
IloPredicate<IloNumVar> pred2 = IsBoundInTheSolution(env, solution);
IloBool bound = pred2(v);
```

For more information, see Selectors.

Macro IloQuarterPi

Definition file: ilconcert/ilosys.h

`IloQuarterPi`

Quarter pi.
Concert Technology predefines conventional trigonometric constants to conform to IEEE 754 standards for quarter pi, half pi, pi, three-halves pi, and two pi.

```
extern const IloNum IloQuarterPi;    // = 0.78539816339744830962
```

Macro ILOSELECTOR0

Definition file: ilsolver/iloselector.h

ILOSELECTOR0(name, tx, tc, nc)
ILOSELECTOR1(name, tx, tc, nc, td, nd)
ILOSELECTOR2(name, tx, tc, nc, td1, nd1, td2, nd2)
ILOSELECTOR3(name, tx, tc, nc, td1, nd1, td2, nd2, td3, nd3)
ILOSELECTOR4(name, tx, tc, nc, td1, nd1, td2, nd2, td3, nd3, td4, nd4)
ILOSELECTOR5(name, tx, tc, nc, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5)
ILOSELECTOR6(name, tx, tc, nc, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5, td6, nd6)
ILOSELECTOR7(name, tx, tc, nc, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5, td6, nd6, td7, nd7)

The `ILOSELECTORi` macros define several functions named `name` that create and return an instance of selector of objects of class `tx` with a selection function using *i* extra fields of type `td1...td`*i*.

The signatures of these functions are:

- `IloSelector<tx,tc> name(IloEnv)`, that returns a selector allocated on the environment; and
- `IloSelector<tx,tc> name(IloSolver)`, that returns a selector allocated on the solver

The user-written code specifies which element of the given collection of type `tc` is selected by invoking the `select(tx)` method.

Example

The following code defines a selector which randomly selects a variable in an array:

```
ILOSELECTOR0(RandomVariableInArraySelector,
             IlcIntVar,
             IlcIntVarArray, array) {
  IlcInt index = array[0].getManager().getRandom().getInt(array.getSize()-1);
  select(array[index]);
}
```

This macro defines the following function:

```
IloSelector<IlcIntVar,IlcIntVarArray> RandomVariableInArraySelector(IloSolver);
```

For example, the following code defines an instance of a selector to select a random variable in an `IlcIntVarArray`:

```
IloSolver solver = ...;
IloSelector<IlcIntVar,IlcIntVarArray> randomVariableInArraySelector = RandomVariableInArraySelector(solver);
IlcIntVarArray array = ...;
IlcIntVar selected;
IlcBool isSelected = randomVariableInArraySelector.select(selected, array);
```

For more information, see Selectors.

Macro ILOSTLBEGIN

Definition file: ilconcert/ilosys.h

ILOSTLBEGIN

Macro for STL.
This macro enables you run your application either with the STL (Standard Template Library) of Microsoft Visual C++ or with other platforms. It is defined as:

```
using namespace std
```

when the STL is used (ports of type `stat_sta`, `stat_mta` or `stat_mda`); otherwise, its value is simply null.

Macro IloThreeHalfPi

Definition file: ilconcert/ilosys.h

IloThreeHalfPi

Three half-pi.
Concert Technology predefines conventional trigonometric constants to conform to IEEE 754 standards for quarter pi, half pi, pi, three-halves pi, and two pi.

```
extern const IloNum IloThreeHalfPi;  // = 4.71238898038468985769
```

Macro ILOTRANSLATOR

Definition file: ilsolver/iloselector.h

ILOTRANSLATOR(name, IloObjectOut, IloObjectIn, nx)

This macro defines a translator class that translates objects of type IloObjectIn into objects of type IloObjectOut. This macro defines a class name which is a subclass of template class IloTranslator<IloObjectOut,IloObjectIn>. The use of this macro is the only way to define a new subclass of translator.

Example

This example shows how to define a translator that translates an instance of IlcIntSetVar into an IlcIntSet that represents the required set of the IlcIntSetVar:

```
ILOTRANSLATOR(RequiredSetTranslator,
              IlcIntSet,
              IlcIntSetVar, v) {
 return v.getRequiredSet();
}
```

This macro defines a class RequiredSetTranslator which is a subclass of IloTranslator<IlcIntSet,IlcIntSetVar> that can be used to transform a predicate or evaluator on IlcIntSet into the corresponding predicate or evaluator on IlcIntSetVar using operator<< as illustrated in the following code:

```
IloTranslator<IlcIntSet,IlcIntSetVar> tr = RequiredSetTranslator(solver);
IloEvaluator<IlcIntSet> setSizeEvaluator = ...;
IloEvaluator<IlcIntSetVar> requiredSetSizeEvaluator = setSizeEvaluator << tr;
```

For more information, see Selectors.

See Also: operator<<, operator<<

Macro IloTwoPi

Definition file: ilconcert/ilosys.h

IloTwoPi

Two pi.
Concert Technology predefines conventional trigonometric constants to conform to IEEE 754 standards for quarter pi, half pi, pi, three-halves pi, and two pi.

```
extern const IloNum IloTwoPi;        // = 6.28318530717958647692
```

Macro ILOVISITOR0

Definition file: ilsolver/iloselector.h

```
ILOVISITOR0(name, tx, tc, nc)
ILOVISITOR1(name, tx, tc, nc, td, nd)
ILOVISITOR2(name, tx, tc, nc, td1, nd1, td2, nd2)
ILOVISITOR3(name, tx, tc, nc, td1, nd1, td2, nd2, td3, nd3)
ILOVISITOR4(name, tx, tc, nc, td1, nd1, td2, nd2, td3, nd3, td4, nd4)
ILOVISITOR5(name, tx, tc, nc, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5)
ILOVISITOR6(name, tx, tc, nc, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5,
td6, nd6)
ILOVISITOR7(name, tx, tc, nc, td1, nd1, td2, nd2, td3, nd3, td4, nd4, td5, nd5,
td6, nd6, td7, nd7)
```

The `ILOVISITORi` macros allow you to define a new visitor of name `name` for a given object class `tx` and a given container class `tc` with *i* data members of type `td1...td`*i*.

This function's signature returns a visitor allocated on an environment or on a solver heap:

```
IloVisitor<tx,tc> name(IloEnv, td1, ..., tdi);
IloVisitor<tx,tc> name(IloSolver, td1, ..., tdi);
```

Within the code of this macro, the function `void visit(tx object)` allows you to specify each visited object whereas the function `IloBool keepVisiting()` returns `IloFalse` if it is not necessary to visit objects anymore.

Here is an example that defines a new visitor `IntVarArrayBackwardVisitor`, an instance of `IloVisitor<IloIntVar,IloIntVarArray>` that visits all the variables of an integer variable array in backward order:

```
ILOVISITOR0(IntVarArrayBackwardVisitor,IloIntVar,IloIntVarArray,array) {
   for (IloInt i=array.getSize()-1; i>=0; --i)
       visit(array[i]);
}
```

For more information, see Selectors.

See Also: IloVisitor, IloBestSelector

Typedef IlcAny

Definition file: ilsolver/ilcerr.h
Include file: <ilsolver/ilosolver.h>

```
IloAny IlcAny
```

This type represents objects handled by Solver enumerated variables and by Solver set variables. A pointer to any object, whether a predefined Solver type or an instance of a C++ class, is implicitly converted to `IlcAny`. By using this type, you can be sure that the Solver components of your application will port without any source change in this respect across different hardware platforms.

See Also: IlcAnyArray, IlcRevAny

Typedef IlcBool

Definition file: ilsolver/ilcerr.h
Include file: <ilsolver/ilosolver.h>

```
IloBool IlcBool
```

This type represents Boolean values in Solver; those values are `IlcTrue` and `IlcFalse`. Booleans are, in fact, integers of the type `IlcInt: IlcFalse` is 0 (zero), and `IlcTrue` is 1 (one). This type anticipates the built-in `bool` type proposed for standard C++. By using this type, you can be sure that the Solver components of your application will port without any source change in this respect across different hardware platforms.

See Also: IlcBoolVar, IlcRevBool

Typedef IlcChooseAnyIndex

Definition file: ilsolver/ilcany.h
Include file: <ilsolver/ilosolver.h>

```
IlcInt(* IlcChooseAnyIndex)(const IlcAnyVarArray)
```

This C++ type represents a pointer to a function that takes an array of constrained enumerated variables and returns an integer.

In its search primitives for finding solutions, Solver lets you set parameters for choosing the order in which constrained variables are bound. You do so by means of choice functions called *criteria*.

The choice functions for constrained enumerated variables (that is, instances of `IlcAnyExp` or its subclasses) should have a signature of this type.

See Also: IlcAnyVarArray, IloBestGenerate, IlcChooseFirstUnboundAny, IlcChooseIndex1, IlcChooseIndex2, IlcChooseMinSizeAny, IloGenerate, IlcBoolVarArray

Typedef IlcChooseAnySetIndex

Definition file: ilsolver/ilcset.h
Include file: <ilsolver/ilosolver.h>

```
IlcInt(* IlcChooseAnySetIndex)(const IlcAnySetVarArray)
```

This C++ type represents a pointer to a function that takes an array of constrained enumerated set variables and returns an integer.

In its search primitives for finding solutions, Solver lets you set parameters for choosing the order in which constrained variables are bound. You do so by means of choice functions called *criteria*.

The choice functions for constrained enumerated set variables (that is, instances of `IlcAnySetVar` or its subclasses) should have a signature of this type.

See Also: IlcAnySetVarArray, IloBestGenerate, IlcChooseFirstUnboundAnySet, IlcChooseIndex1, IlcChooseIndex2, IlcChooseMinSizeAnySet, IloGenerate

Typedef IlcChooseFloatIndex

Definition file: ilsolver/linfloat.h
Include file: <ilsolver/ilosolver.h>

```
IlcInt(* IlcChooseFloatIndex)(const IlcFloatVarArray)
```

This C++ type represents a pointer to a function that takes an array of constrained floating-point variables and returns an integer.

In its search primitives for finding solutions, Solver lets you set parameters for choosing the order in which constrained variables are bound. You do so by means of choice functions called *criteria*.

The choice functions for constrained floating-point expressions (that is, instances of `IlcFloatExp` or its subclasses) should have a signature of this type.

See Also: IloBestGenerate, IlcChooseFloatIndex1, IlcChooseFloatIndex2, IlcChooseFirstUnboundFloat, IlcChooseMaxMaxFloat, IlcChooseMaxMinFloat, IlcChooseMaxSizeFloat, IlcChooseMinMaxFloat, IlcChooseMinMinFloat, IlcChooseMinSizeFloat, IlcFloatVarArray, IloGenerate

Typedef IlcChooseIntIndex

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

```
IlcInt(* IlcChooseIntIndex)(const IlcIntVarArray)
```

This C++ type represents a pointer to a function that takes an array of constrained integer expressions and returns an integer.

In its search primitives for finding solutions, Solver lets you set parameters for choosing the order in which constrained variables are bound. You do so by means of choice functions called *criteria*.

The choice functions for constrained integer variables (that is, instances of `IlcIntExp` or its subclasses) should have a signature of this type.

See Also: IloBestGenerate, IlcChooseFirstUnboundInt, IlcChooseIndex1, IlcChooseIndex2, IlcChooseIntIndex, IlcChooseMaxMaxInt, IlcChooseMaxMinInt, IlcChooseMaxSizeInt, IlcChooseMaxRegretMax, IlcChooseMaxRegretMin, IlcChooseMinMaxInt, IlcChooseMinMinInt, IlcChooseMinRegretMax, IlcChooseMinRegretMin, IlcChooseMinSizeInt, IloGenerate, IlcIntVarArray

Typedef IlcChooseIntSetIndex

Definition file: ilsolver/intset.h
Include file: <ilsolver/ilosolver.h>

```
IlcInt(* IlcChooseIntSetIndex)(const IlcIntSetVarArray)
```

This C++ type represents a pointer to a function that takes an array of constrained integer set variables and returns an integer.

In its search primitives for finding solutions, Solver lets you set parameters for choosing the order in which constrained variables are bound. You do so by means of choice functions called *criteria*.

The choice functions for constrained integer set variables (that is, instances of `IlcIntSetVar` or its subclasses) should have a signature of this type.

See Also: IloBestGenerate, IlcChooseFirstUnboundIntSet, IlcChooseIndex1, IlcChooseIndex2, IlcChooseMinSizeIntSet, IloGenerate, IlcIntSetVarArray

Typedef IlcEvalAny

Definition file: ilsolver/ilcany.h
Include file: <ilsolver/ilosolver.h>

```
IlcInt(* IlcEvalAny)(const IlcAny val, IlcAnyVar var)
```

This C++ type represents a pointer to a function that takes two arguments and returns an integer. The second of those two arguments is a constrained enumerated variable (that is, an instance of `IlcAnyExp` or one of its subclasses), and the first argument is a pointer.

Solver lets you control the order in which the values in the domain of a constrained variable are tried during the search for a solution. The choice of the next value to try for a constrained enumerated variable is made by an object of the class `IlcAnySelect`. Such an object usually uses an evaluation function of the type `IlcEvalAny`.

See Also: IlcAnySelect, IlcAnyVar, IloInstantiate

Typedef IlcEvalAnySet

Definition file: ilsolver/ilcset.h
Include file: <ilsolver/ilosolver.h>

```
IlcInt(* IlcEvalAnySet)(IlcAny val, IlcAnySetVar var)
```

This C++ type represents a pointer to a function that takes two arguments and returns an integer. The second of those two arguments is a constrained enumerated set variable (that is, an instance of `IlcAnySetVar` or one of its subclasses), and the first argument is a pointer.

Solver lets you control the order in which the values in the domain of a constrained variable are tried during the search for a solution. The choice of the next value to try for a constrained enumerated set variable is made by an object of the class `IlcAnySetSelect`. Such an object usually uses an evaluation function of the type `IlcEvalAnySet`.

See Also: IlcAnySetSelect, IlcAnySetVar, IloInstantiate

Typedef IlcEvalInt

Definition file: ilsolver/ilcint.h
Include file: <ilsolver/ilosolver.h>

```
IlcInt(* IlcEvalInt)(IlcInt val, IlcIntVar var)
```

This C++ type represents a pointer to a function that takes two arguments and returns an integer. The second of those two arguments is a constrained integer variable (that is, an instance of `IlcIntExp` or one of its subclasses), and the first argument is an integer.

Solver lets you control the order in which the values in the domain of a constrained variable are tried during the search for a solution. The choice of the next value to try for a constrained integer variable is made by an object of the class `IlcIntSelect`. Such an object usually uses an evaluation function of the type `IlcEvalInt`.

See Also: IloBestInstantiate, IloInstantiate, IlcIntSelect, IlcIntSelectEvalI, IlcIntSelectI

Typedef IlcEvalIntSet

Definition file: ilsolver/intset.h
Include file: <ilsolver/ilosolver.h>

```
IlcInt(* IlcEvalIntSet)(IlcInt val, IlcIntSetVar var)
```

This C++ type represents a pointer to a function that takes two arguments and returns an integer. The second of those two arguments is a constrained integer set variable (that is, an instance of `IlcIntSetVar` or one of its subclasses), and the first argument is an integer.

Solver lets you control the order in which the values in the domain of a constrained variable are tried during the search for a solution. The choice of the next value to try for a constrained integer set variable is made by an object of the class `IlcIntSetSelect`. Such an object usually uses an evaluation function of the type `IlcEvalIntSet`.

See Also: IlcIntSetSelect, IlcIntSetVar, IloInstantiate

Typedef IlcFloat

Definition file: ilsolver/ilcerr.h
Include file: <ilsolver/ilosolver.h>

```
IloNum IlcFloat
```

This type represents floating-point numbers handled by Solver floating-point variables. This type corresponds to the type `double`. By using this type, you can be sure that the Solver components of your application will port without any source change in this respect across different hardware platforms.

IEEE 754 is a standard proposed by the Institute of Electronic and Electrical Engineers for computing floating-point arithmetic. The implementation of floating-point numbers in Solver conforms to this standard. See the *IBM ILOG Solver User's Manual* for a discussion of floating-point arithmetic.

See Also: IlcComputeMax, IlcComputeMin, IlcFloatArray, IlcFloatMax, IlcFloatMin

Typedef IlcFloatFunction

Definition file: ilsolver/flti.h
Include file: <ilsolver/ilosolver.h>

```
IlcFloat(* IlcFloatFunction)(IlcFloat)
```

This C++ type represents a pointer to a function that takes an argument of type `IlcFloat` and returns an object of type `IlcFloat`. This type is the type of the function passed as a parameter to the function `IlcMonotonicIncreasingFloatExp`.

See Also: IlcMonotonicIncreasingFloatExp, IlcMonotonicDecreasingFloatExp

Typedef IlcFloatVarRef

Definition file: ilsolver/search.h
Include file: <ilsolver/ilosolver.h>

```
IlcFloatExpI ** IlcFloatVarRef
```

This type definition creates a reference to a floating-point variable. A node evaluator, such as an instance of `IlcMTBFSEvaluator`, might use this type definition.

See Also: IlcMTBFSEvaluator

Typedef IlcInt

Definition file: ilsolver/ilcerr.h
Include file: <ilsolver/ilosolver.h>

```
IloInt IlcInt
```

This type represents signed integers handled by Solver variables. All C++ integers are implicitly converted to `IlcInt`. By using this type, you can be sure that the Solver components of your application will port without any source change in this respect across different hardware platforms.

See Also: IlcIntArray, IlcIntMax, IlcIntMin, IlcRevInt

Typedef IlcIntVarRef

Definition file: ilsolver/search.h
Include file: <ilsolver/ilosolver.h>

```
IlcIntExpI ** IlcIntVarRef
```

This type definition creates a reference to an integer variable. A node evaluator, such as an instance of `IlcMTBFSEvaluator`, might use this type definition.

See Also: IlcMTBFSEvaluator

Typedef IlcPathTransitFunction

Definition file: ilsolver/ilcpath.h
Include file: <ilsolver/ilosolver.h>

```
IlcFloat(* IlcPathTransitFunction)(IlcInt, IlcInt)
```

This C++ type represents a pointer to a function that takes two arguments and returns a floating-point number. The two arguments are the indices of nodes. The function should return a *transit cost* for connecting the two nodes. This transit cost can be the distance between the nodes or the cost of a path visiting either of the nodes. This kind of function is known as a *transit function*.

See Also: IlcPathLength, IlcPathTransit, IlcPathTransitEvalI, IlcPathTransitI

Typedef IloAny

Definition file: ilconcert/ilosys.h

```
void *ILO_MAY_ALIAS IloAny
```

For constraint programming: the type for objects as variables in enumerations or sets.
This type definition represents objects in a model handled by Concert Technology enumerated variables and by Concert Technology set variables. A pointer to any object, whether a predefined Concert Technology type or an instance of a C++ class, is implicitly converted to `IloAny`. By using this type, you can be sure that these components of your application will port without any source change across different hardware platforms.

See Also: IloModel

Typedef IloBool

Definition file: ilconcert/ilosys.h

```
IloInt IloBool
```

Type for Boolean values.
This type definition represents Boolean values in Concert Technology. Those values are `IloTrue` and `IloFalse`. Booleans are, in fact, integers of type `IloInt`. `IloFalse` is 0 (zero), and `IloTrue` is 1 (one). This type anticipates the built-in `bool` type proposed for standard C++. By using this type, you can be sure that the Concert Technology components of your application will port in this respect without source changes across different hardware platforms.

See Also: IloBoolArray, IloInt, IloModel, IloNum

Typedef IloInt

Definition file: ilconcert/ilosys.h

```
long IloInt
```

Type for signed integers.
This type definition represents signed integers in Concert Technology.

See Also: IloBool, IloModel, IloNum

Typedef IloNum

Definition file: ilconcert/ilosys.h

```
double IloNum
```

Type for numeric values as floating-point numbers.
This type definition represents numeric values as floating-point numbers in Concert Technology.

See Also: IloModel, IloInt

Typedef IloNumFunction

Definition file: ilconcert/ilosys.h

```
IloNum(* IloNumFunction)(IloNum)
```

For IBM® ILOG® Solver: the type for a pointer to a numeric function.
This C++ type represents a pointer to a function that takes an argument of type `IloNum` and returns an object of type `IloNum`. This type is the type of the function passed as an argument to the function `IloMonotonicDecreasingNumExpr`.

See Also: IloMonotonicDecreasingNumExpr, IloMonotonicIncreasingNumExpr

Typedef IloPathTransitFunction

Definition file: ilconcert/ilomodel.h

```
IloNum(* IloPathTransitFunction)(IloInt i, IloInt j)
```

For IBM® ILOG® Solver: a pointer to a function that computes a transit cost of connecting two nodes.
This C++ type definition represents a pointer to a function that takes two arguments and returns a floating-point number. The two arguments are the indices of nodes. The function should return a *transit cost* for connecting the two nodes. This transit cost can be the distance between the nodes or the cost of a path visiting either of the nodes. This kind of function is known as a *transit function*.

See Also: IloPathLength, IloPathTransitI

Typedef IloPoolProcArray

Definition file: ilsolver/iimiloproc.h

```
IloArray< IloPoolProc > IloPoolProcArray
```

An array of pool processors.
This typedef creates an array of pool processors. This C++ typedef is provided for convenience.

> **Note**
>
> IIM provides a default visitor for this type. Thus, selection of a pool processor from an array can be performed without specification of a visitor.

See Also: IloVisitor, IloSelector, IloBestSelector

Typedef IloSolutionArray

Definition file: ilconcert/ilosolution.h

```
IloSimpleArray< IloSolution > IloSolutionArray
```

Type definition for arrays of `IloSolution` instances.
This type definition represents arrays of instances of `IloSolution`.

Instances of `IloSolutionArray` are extensible. That is, you can add more elements to such an array. References to an array change whenever an element is added or removed from the array.

See Also: IloSolution

Typedef IloSolutionPoolEvaluator

Definition file: ilsolver/iimmulti.h
Include file: <ilsolver/iim.h>

```
IloMultipleEvaluator< IloSolution, IloSolutionPool > IloSolutionPoolEvaluator
```

An explicit evaluator of solution pools.
This typedef allows you to write less code when you heavily use IIM selector classes.

Typedef IloSolutionPoolSelector

Definition file: ilsolver/iimiloproc.h
Include file: <ilsolver/iim.h>

```
IloSelector< IloSolution, IloSolutionPool > IloSolutionPoolSelector
```

A selector which selects solutions from pools.
This typedef allows you to write less code when you heavily use IIM selector classes.

Variable ILC_NO_MEMORY_MANAGER

Definition file: ilsolver/basic.h
Include file: <ilsolver/ilosolver.h>
This operating-system environment variable enables you to control the memory manager of Solver. It replaces `IloSolver::useHeap`. Solver uses its own memory manager to provide faster memory allocation for certain Solver objects. The use of this memory manager can hide memory problems normally detected by memory usage applications (such as Rational Purify, for example). If you are working in a software development environment capable of detecting bad memory access, you can use this operating-system environment variable to turn off the Solver memory manager in order to detect such anomalies during software development. This environment variable also applies to reversible memory. For example, if you are working in such a development environment on a personal computer running Microsoft NT, use this statement:

```
set ILC_NO_MEMORY_MANAGER=1
```

If you are working on a UNIX platform, using a C-shell, here is one way of setting this environment variable:

```
setenv ILC_NO_MEMORY_MANAGER
```

Variable IlcFloatMax

Definition file: ilsolver/flti.h
Include file: <ilsolver/ilosolver.h>
This global floating-point constant is equal to the highest floating-point number according to IEEE 754.

IEEE 754 is a standard proposed by the Institute of Electronic and Electrical Engineers for computing floating-point arithmetic. The implementation of floating-point numbers in Solver conforms to this standard. See the *IBM ILOG Solver User's Manual* for a discussion of floating-point arithmetic.

See Also: IlcComputeMax, IlcComputeMin, IlcFloat, IlcFloatMin, IlcInfinity

Variable IlcFloatMin

Definition file: ilsolver/linfloat.h
Include file: <ilsolver/ilosolver.h>
This global floating-point constant is equal to the smallest, strictly positive, normalized floating-point number according to IEEE 754.

IEEE 754 is a standard proposed by the Institute of Electronic and Electrical Engineers for computing floating-point arithmetic. The implementation of floating-point numbers in Solver conforms to this standard. See the *IBM ILOG Solver User's Manual* for a discussion of floating-point arithmetic.

See Also: IlcComputeMax, IlcComputeMin, IlcFloat, IlcFloatMax, IlcInfinity

Variable ILO_NO_MEMORY_MANAGER

Definition file: ilconcert/ilosys.h
OS environment variable controls Concert Technology memory manager.
This operating-system environment variable enables you to control the memory manager of Concert Technology.

Concert Technology uses its own memory manager to provide faster memory allocation for certain Concert Technology objects. The use of this memory manager can hide memory problems normally detected by memory usage applications (such as Rational Purify, for example). If you are working in a software development environment capable of detecting bad memory access, you can use this operating-system environment variable to turn off the Concert Technology memory manager in order to detect such anomalies during software development.

For example, if you are working in such a development environment on a personal computer running Microsoft XP, use this statement:

```
set ILO_NO_MEMORY_MANAGER=1
```

If you are working on a UNIX platform, using a C-shell, here is one way of setting this environment variable:

```
setenv ILO_NO_MEMORY_MANAGER
```

Variable IloInfinity

Definition file: ilconcert/ilosys.h
Largest double-precision floating-point number.
This symbolic constant represents the largest double-precision floating-point number on a given platform. It is initialized when you create an instance of `IloEnv`. In practice, when you use this symbolic constant as an upper bound of a variable in your model, you are effectively stating that the variable is unbounded.

See the *IBM ILOG Solver Reference Manual* and *User's Manual* for details about how IBM® ILOG® Solver treats floating-point calculations in instances of `IloSolver` in conformity with IEEE 754. In particular, IBM ILOG Solver offers other symbolic constants, such as `IlcIntMax` or `IlcFloatMax` that may be more appropriate for your application if you do not intend to state that your variables are effectively unbounded.

See the IBM ILOG CPLEX Reference Manual and User's Manual for details about how IBM ILOG CPLEX treats floating-point calculations in instances of `IloCplex`.

See Also: IloEnv

Variable IloIntMax

Definition file: ilconcert/ilosys.h
Largest integer.
These symbolic constants represent the largest integer on a given platform.

Variable IloIntMin

Definition file: ilconcert/ilosys.h
Least integer.
These symbolic constants represent the least integer on a given platform.