IBM XL C/C++ Advanced Edition for Linux, V9.0

Programming Guide

SC23-5890-00

IBM XL C/C++ Advanced Edition for Linux, V9.0

Programming Guide

SC23-5890-00

Note!
Before using this information and the product it supports, be sure to read the general information under "Notices" on page 81.

First Edition

This edition applies to IBM XL C/C++ Advanced Edition for Linux, V9.0 (Program number 5724-S73) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Contents

About this document

This guide discusses advanced topics related to the use of the IBM® XL C/C++ Advanced Edition for Linux®, V9.0 compiler, with a particular focus on program portability and optimization. The guide provides both reference information and practical tips for getting the most out of the compiler's capabilities, through recommended programming practices and compilation procedures.

Who should read this document

This document is addressed to programmers building complex applications, who already have experience compiling with XL C/C++, and would like to take further advantage of the compiler's capabilities for program optimization and tuning, support for advanced programming language features, and add-on tools and utilities.

How to use this document

This document uses a ″task-oriented″ approach to presenting the topics, by concentrating on a specific programming or compilation problem in each section. Each topic contains extensive cross-references to the relevant sections of the reference guides in the IBM XL C/C++ Advanced Edition for Linux, V9.0 documentation set, which provide detailed descriptions of compiler options and pragmas, and specific language extensions.

How this document is organized

This guide includes these topics:

- Chapter 1, "Using 32-bit and 64-bit modes ," on page 1 discusses common problems that arise when porting existing 32-bit applications to 64-bit mode, and provides recommendations for avoiding these problems.
- Chapter 2, "Using XL C/C++ with Fortran," on page 5 discussions considerations for calling Fortran code from XL C/C++ programs.
- Chapter 3, "Aligning data," on page 9 discusses the different compiler options available for controlling the alignment of data in aggregates, such as structures and classes, on all platforms.
- Chapter 4, "Handling floating point operations," on page 17 discusses options available for controlling the way floating-point operations are handled by the compiler.
- Chapter 5, "Using C++ templates," on page 21 discusses the different options for compiling programs that include C++ templates.
- Chapter 6, "Constructing a library," on page 27 discusses how to compile and link static and shared libraries, and how to specify the initialization order of static objects in C++ programs.
- Chapter 7, "Optimizing your applications," on page 33 discusses the various options provided by the compiler for optimizing your programs, and provides recommendations for use of the different options.
- Chapter 9, "Coding your application to improve performance," on page 57 discusses recommended programming practices and coding techniques for enhancing program performance and compatibility with the compiler's optimization capabilities.

- Chapter 10, “Using the high performance libraries,” on page 63 discusses two performance libraries that are shipped with XL C/C++: the Mathematical Acceleration Subsystem (MASS), which contains tuned versions of standard math library functions; and the Basic Linear Algebra Subprograms (BLAS), which contains basic functions for matrix multiplication.
- Chapter 11, “Parallelizing your programs,” on page 75 provides an overview of the different options offered by the IBM XL C/C++ Advanced Edition for Linux, V9.0 for creating multi-threaded programs, including OpenMP language constructs.

Conventions used in this document

Typographical conventions

The following table explains the typographical conventions used in this document.

Table 1. Typographical conventions

Typeface	Indicates	Example
bold	Lowercase commands, executable names, compiler options and directives.	If you specify **-O3**, the compiler assumes **-qhot=level=0**. To prevent all HOT optimizations with **-O3**, you must specify **-qnohot**.
italics	Parameters or variables whose actual names or values are to be supplied by the user. Italics are also used to introduce new terms.	Make sure that you update the *size* parameter if you return more than the *size* requested.
`monospace`	Programming keywords and library functions, compiler built-in functions, examples of program code, command strings, or user-defined names.	If one or two cases of a `switch` statement are typically executed much more frequently than other cases, break out those cases by handling them separately before the `switch` statement.

Icons

All features described in this document apply to both C and C++ languages. Where a feature is exclusive to one language, or where functionality differs between languages, the following icons are used:

The text describes a feature that is supported in the C language only; or describes behavior that is specific to the C language.

The text describes a feature that is supported in the C++ language only; or describes behavior that is specific to the C++ language.

Syntax diagrams

Throughout this document, diagrams illustrate XL C/C++ syntax. This section will help you to interpret and use those diagrams.

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ►►— symbol indicates the beginning of a command, directive, or statement.

The ⟶ symbol indicates that the command, directive, or statement syntax is continued on the next line.

The ►— symbol indicates that a command, directive, or statement is continued from the previous line.

The ⟶◄ symbol indicates the end of a command, directive, or statement.

Fragments, which are diagrams of syntactical units other than complete commands, directives, or statements, start with the |— symbol and end with the —| symbol.

- Required items are shown on the horizontal line (the main path):

- Optional items are shown below the main path:

- If you can choose from two or more items, they are shown vertically, in a stack.

  If you *must* choose one of the items, one item of the stack is shown on the main path.

  If choosing one of the items is optional, the entire stack is shown below the main path.

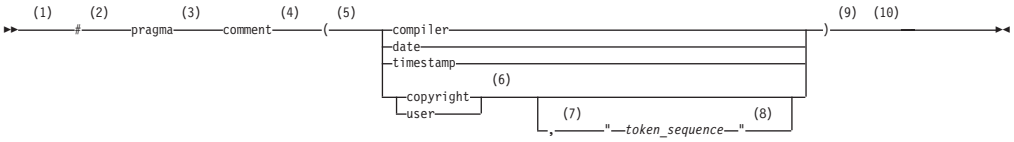
- An arrow returning to the left above the main line (a repeat arrow) indicates that you can make more than one choice from the stacked items or repeat an item. The separator character, if it is other than a blank, is also indicated:

- The item that is the default is shown above the main path.

- Keywords are shown in nonitalic letters and should be entered exactly as shown.
- Variables are shown in italicized lowercase letters. They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Sample syntax diagram

The following syntax diagram example shows the syntax for the **#pragma comment** directive.



Notes:

1 This is the start of the syntax diagram.

2 The symbol # must appear first.

3 The keyword `pragma` must appear following the # symbol.

4 The name of the pragma `comment` must appear following the keyword `pragma`.

5 An opening parenthesis must be present.

6 The comment type must be entered only as one of the types indicated: `compiler`, `date`, `timestamp`, `copyright`, or `user`.

7 A comma must appear between the comment type `copyright` or `user`, and an optional character string.

8 A character string must follow the comma. The character string must be enclosed in double quotation marks.

9 A closing parenthesis is required.

10 This is the end of the syntax diagram.

The following examples of the **#pragma comment** directive are syntactically correct according to the diagram shown above:

```
#pragma
comment(date)
#pragma comment(user)
#pragma comment(copyright,"This text will appear in the module")
```

Examples

The examples in this document, except where otherwise noted, are coded in a simple style that does not try to conserve storage, check for errors, achieve fast performance, or demonstrate all possible methods to achieve a specific result.

Related information

The following sections provide information on documentation related to XL C/C++:

IBM XL C/C++ publications

XL C/C++ provides product documentation in the following formats:

- README files

README files contain late-breaking information, including changes and corrections to the product documentation. README files are located by default in the XL C/C++ directory and in the root directory of the installation CD.

- Installable man pages

  Man pages are provided for the compiler invocations and all command-line utilities provided with the product. Instructions for installing and accessing the man pages are provided in the *XL C/C++ Installation Guide*.

- Information center

  The information center of searchable HTML files can be launched on a network and accessed remotely or locally. Instructions for installing and accessing the online information center are provided in the *XL C/C++ Installation Guide*. The information center is also viewable on the Web at http://publib.boulder.ibm.com/infocenter/lnxphelp/v9v111/index.jsp.

- PDF documents

  PDF documents are located by default in the /opt/ibmcmp/vac/9.0/doc/*LANG*/pdf/ directory, where *LANG* is one of en_US, zh_CN, or ja_JP. The PDF files are also available on the Web at http://www.ibm.com/software/awdtools/xlcpp/library.

  The following files comprise the full set of XL C/C++ product manuals:

Table 2. XL C/C++ PDF files

Document title	PDF file name	Description
IBM XL C/C++ Advanced Edition for Linux, V9.0 Installation Guide, GC23-5893-00	install.pdf	Contains information for installing XL C/C++ and configuring your environment for basic compilation and program execution.
Getting Started with IBM XL C/C++ Advanced Edition for Linux, V9.0, GC23-5891-00	getstart.pdf	Contains an introduction to the XL C/C++ product, with information on setting up and configuring your environment, compiling and linking programs, and troubleshooting compilation errors.
IBM XL C/C++ Advanced Edition for Linux, V9.0 Compiler Reference, SC23-5889-00	compiler.pdf	Contains information about the various compiler options, pragmas, macros, environment variables, and built-in functions, including those used for parallel processing.
IBM XL C/C++ Advanced Edition for Linux, V9.0 Language Reference, SC23-5892-00	langref.pdf	Contains information about the C and C++ programming languages, as supported by IBM, including language extensions for portability and conformance to non-proprietary standards.
IBM XL C/C++ Advanced Edition for Linux, V9.0 Programming Guide, SC23-5890-00	proguide.pdf	Contains information on advanced programming topics, such as application porting, interlanguage calls with Fortran code, library development, application optimization and parallelization, and the XL C/C++ high-performance libraries.

To read a PDF file, use the Adobe® Reader. If you do not have the Adobe Reader, you can download it (subject to license terms) from the Adobe Web site at http://www.adobe.com.

More documentation related to XL C/C++ including redbooks, white papers, tutorials, and other articles, is available on the Web at:

http://www.ibm.com/software/awdtools/xlcpp/library

Standards and specifications documents

XL C/C++ is designed to support the following standards and specifications. You can refer to these standards for precise definitions of some of the features found in this document.

- *Information Technology – Programming languages – C, ISO/IEC 9899:1990*, also known as *C89*.
- *Information Technology – Programming languages – C, ISO/IEC 9899:1999*, also known as *C99*.
- *Information Technology – Programming languages – C++, ISO/IEC 14882:1998*, also known as *C++98*.
- *Information Technology – Programming languages – C++, ISO/IEC 14882:2003(E)*, also known as *Standard C++*.
- *Information Technology – Programming languages – Extensions for the programming language C to support new character data types, ISO/IEC DTR 19769*. This draft technical report has been accepted by the C standards committee, and is available at http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1040.pdf.
- *Draft Technical Report on C++ Library Extensions, ISO/IEC DTR 19768*. This draft technical report has been submitted to the C++ standards committee, and is available at http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2005/n1836.pdf.
- *AltiVec Technology Programming Interface Manual*, Motorola Inc. This specification for vector data types, to support vector processing technology, is available at http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf.
- *OpenMP Application Program Interface Version 2.5*, available at http://www.openmp.org

Other IBM publications

- *ESSL for Linux on POWER V4.2 Guide and Reference*, SA22-7904, available at http://publib.boulder.ibm.com/infocenter/clresctr/index.jsp

Other publications

- *Using the GNU Compiler Collection* available at http://gcc.gnu.org/onlinedocs

Technical support

Additional technical support is available from the XL C/C++ Support page at http://www.ibm.com/software/awdtools/xlcpp/support. This page provides a portal with search capabilities to a large selection of technical support FAQs and other support documents.

If you cannot find what you need, you can send e-mail to compinfo@ca.ibm.com.

For the latest information about XL C/C++, visit the product information site at http://www.ibm.com/software/awdtools/xlcpp.

How to send your comments

Your feedback is important in helping to provide accurate and high-quality information. If you have any comments about this document or any other XL C/C++ documentation, send your comments by e-mail to compinfo@ca.ibm.com.

Be sure to include the name of the document, the part number of the document, the version of XL C/C++, and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).

Chapter 1. Using 32-bit and 64-bit modes

You can use XL C/C++ to develop both 32-bit and 64-bit applications. To do so, specify **-q32** (the default) or **-q64**, respectively, during compilation.

However, porting existing applications from 32-bit to 64-bit mode can lead to a number of problems, mostly related to the differences in C/C++ long and pointer data type sizes and alignment between the two modes. The following table summarizes these differences.

Table 3. Size and alignment of data types in 32-bit and 64-bit modes

Data type	**32-bit mode**		**64-bit mode**	
	Size	**Alignment**	**Size**	**Alignment**
long, unsigned long	4 bytes	4-byte boundaries	8 bytes	8-byte boundaries
pointer	4 bytes	4-byte boundaries	8 bytes	8-byte boundaries
size_t (system-defined unsigned long)	4 bytes	4-byte boundaries	8 bytes	8-byte boundaries
ptrdiff_t (system-defined long)	4 bytes	4-byte boundaries	8 bytes	8-byte boundaries

The following sections discuss some of the common pitfalls implied by these differences, as well as recommended programming practices to help you avoid most of these issues:

- "Assigning long values" on page 2
- "Assigning pointers" on page 3
- "Aligning aggregate data" on page 4
- "Calling Fortran code" on page 4

When compiling in 32-bit or 64-bit mode, you can use the **-qwarn64** option to help diagnose some issues related to porting applications. In either mode, the compiler immediately issues a warning if undesirable results, such as truncation or data loss, have occurred.

For suggestions on improving performance in 64-bit mode, see "Optimize operations in 64-bit mode " on page 61.

Related information

- **-q32**/**-q64** and **-qwarn64** in *XL C/C++ Compiler Reference*

Assigning long values

The limits of `long` type integers defined in the `limits.h` standard library header file are different in 32-bit and 64-bit modes, as shown in the following table.

Table 4. Constant limits of long integers in 32-bit and 64-bit modes

Symbolic constant	Mode	Value	Hexadecimal	Decimal
LONG_MIN (smallest signed long)	32-bit	$-(2^{31})$	0x80000000L	–2,147,483,648
	64-bit	$-(2^{63})$	0x8000000000000000L	–9,223,372,036,854,775,808
LONG_MAX (longest signed long)	32-bit	$2^{31}-1$	0x7FFFFFFFL	+2,147,483,647
	64-bit	$2^{63}-1$	0x7FFFFFFFFFFFFFFFL	+9,223,372,036,854,775,807
ULONG_MAX (longest unsigned long)	32-bit	$2^{32}-1$	0xFFFFFFFFUL	+4,294,967,295
	64-bit	$2^{64}-1$	0xFFFFFFFFFFFFFFFFUL	+18,446,744,073,709,551,615

Implications of these differences are:

- Assigning a long value to a `double` variable can cause loss of accuracy.
- Assigning constant values to long-type variables can lead to unexpected results. This issue is explored in more detail in "Assigning constant values to long variables."
- Bit-shifting long values will produce different results, as described in "Bit-shifting long values" on page 3.
- Using `int` and `long` types interchangeably in expressions will lead to implicit conversion through promotions, demotions, assignments, and argument passing, and can result in truncation of significant digits, sign shifting, or unexpected results, without warning.

In situations where a long-type value can overflow when assigned to other variables or passed to functions, you must:

- Avoid implicit type conversion by using explicit type casting to change types.
- Ensure that all functions that return long types are properly prototyped.
- Ensure that long parameters can be accepted by the functions to which they are being passed.

Assigning constant values to long variables

Although type identification of constants follows explicit rules in C and C++, many programs use hexadecimal or unsuffixed constants as ″typeless″ variables and rely on a two's complement representation to exceed the limits permitted on a 32-bit system. As these large values are likely to be extended into a 64-bit `long` type in 64-bit mode, unexpected results can occur, generally at boundary areas such as:

- constant >= `UINT_MAX`
- constant < `INT_MIN`
- constant > `INT_MAX`

Some examples of unexpected boundary side effects are listed in the following table.

Table 5. Unexpected boundary results of constants assigned to long types

Constant assigned to long	Equivalent value	32 bit mode	64 bit mode
–2,147,483,649	INT_MIN–1	+2,147,483,647	–2,147,483,649
+2,147,483,648	INT_MAX+1	–2,147,483,648	+2,147,483,648
+4,294,967,726	UINT_MAX+1	0	+4,294,967,296
0xFFFFFFFF	UINT_MAX	–1	+4,294,967,295
0x100000000	UINT_MAX+1	0	+4,294,967,296
0xFFFFFFFFFFFFFFFF	ULONG_MAX	–1	–1

Unsuffixed constants can lead to type ambiguities that can affect other parts of your program, such as when the results of `sizeof` operations are assigned to variables. For example, in 32-bit mode, the compiler types a number like 4294967295 (`UINT_MAX`) as an unsigned long and `sizeof` returns 4 bytes. In 64-bit mode, this same number becomes a signed long and `sizeof` will return 8 bytes. Similar problems occur when passing constants directly to functions.

You can avoid these problems by using the suffixes `L` (for long constants) or `UL` (for unsigned long constants) to explicitly type all constants that have the potential of affecting assignment or expression evaluation in other parts of your program. In the example cited above, suffixing the number as `4294967295U` forces the compiler to always recognize the constant as an `unsigned int` in 32-bit or 64-bit mode.

Bit-shifting long values

Left-bit-shifting long values will produce different results in 32-bit and 64-bit modes. The examples in the table below show the effects of performing a bit-shift on long constants, using the following code segment:

```
long l=valueL<<1;
```

Table 6. Results of bit-shifting long values

Initial value	Symbolic constant	Value after bit shift	
		32-bit mode	64-bit mode
0x7FFFFFFFL	INT_MAX	0xFFFFFFFE	0x00000000FFFFFFFE
0x80000000L	INT_MIN	0x00000000	0x0000000100000000
0xFFFFFFFFL	UINT_MAX	0xFFFFFFFE	0x1FFFFFFFE

Assigning pointers

In 64-bit mode, pointers and `int` types are no longer the same size. The implications of this are:

- Exchanging pointers and `int` types causes segmentation faults.
- Passing pointers to a function expecting an `int` type results in truncation.
- Functions that return a pointer, but are not explicitly prototyped as such, return an `int` instead and truncate the resulting pointer, as illustrated in the following example.

Although code constructs such as the following are valid in 32-bit mode:

```
a=(char*) calloc(25);
```

Without a function prototype for `calloc`, when the same code is compiled in 64-bit mode, the compiler assumes the function returns an `int`, so `a` is silently truncated, and then sign-extended. Type casting the result will not prevent the truncation, as the address of the memory allocated by `calloc` was already truncated during the return. In this example, the correct solution would be to include the header file, `stdlib.h`, which contains the prototype for `calloc`.

To avoid these types of problems:

- Prototype any functions that return a pointer.
- Be sure that the type of parameter you are passing in a function (pointer or `int`) call matches the type expected by the function being called.
- For applications that treat pointers as an integer type, use type `long` or `unsigned long` in either 32-bit or 64-bit mode.

Aligning aggregate data

Structures are aligned according to the most strictly aligned member in both 32-bit and 64-bit modes. However, since long types and pointers change size and alignment in 64-bit, the alignment of a structure's strictest member can change, resulting in changes to the alignment of the structure itself.

Structures that contain pointers or long types cannot be shared between 32-bit and 64-bit applications. Unions that attempt to share `long` and `int` types, or overlay pointers onto `int` types can change the alignment. In general, you should check all but the simplest structures for alignment and size dependencies.

For detailed information on aligning data structures, including structures that contain bit fields, see Chapter 3, "Aligning data," on page 9.

Calling Fortran code

A significant number of applications use C, C++, and Fortran together, by calling each other or sharing files. It is currently easier to modify data sizes and types on the C side than the on Fortran side of such applications. The following table lists C and C++ types and the equivalent Fortran types in the different modes.

Table 7. Equivalent C/C++ and Fortran data types

C/C++ type	**Fortran type**	
	32-bit	**64-bit**
signed int	INTEGER	INTEGER
signed long	INTEGER	INTEGER*8
unsigned long	LOGICAL	LOGICAL*8
pointer	INTEGER	INTEGER*8
		integer POINTER (8 bytes)

Related information

- Chapter 2, "Using XL C/C++ with Fortran," on page 5

Chapter 2. Using XL C/C++ with Fortran

With XL C/C++, you can call functions written in Fortran from your C and C++ programs. This section discusses some programming considerations for calling Fortran code, in the following areas:

- "Identifiers"
- "Corresponding data types"
- "Character and aggregate data" on page 6
- "Function calls and parameter passing" on page 7
- "Pointers to functions" on page 7
- "Sample program: C/C++ calling Fortran" on page 7 provides an example of a C program which calls a Fortran subroutine.

Related information

- "Calling Fortran code" on page 4

Identifiers

You should follow these recommendations when writing C and C++ code to call functions written in Fortran:

- Avoid using uppercase letters in identifiers. Although XL Fortran folds external identifiers to lowercase by default, the Fortran compiler can be set to distinguish external names by case.
- Avoid using long identifier names. The maximum number of significant characters in XL Fortran identifiers is 250[1].

Corresponding data types

The following table shows the correspondence between the data types available in C/C+ and Fortran. Several data types in C have no equivalent representation in Fortran. Do not use them when programming for interlanguage calls.

Table 8. Correspondence of data types among C, C++ and Fortran

C and C++ data types	Fortran data types
bool (C++)_Bool (C)	LOGICAL(1)
char	CHARACTER
signed char	INTEGER*1
unsigned char	LOGICAL*1
signed short int	INTEGER*2
unsigned short int	LOGICAL*2
signed long int	INTEGER*4
unsigned long int	LOGICAL*4
signed long long int	INTEGER*8

1. The Fortran 90 and 95 language standards require identifiers to be no more than 31 characters; the Fortran 2003 standard requires identifiers to be no more than 63 characters.

Table 8. Correspondence of data types among C, C++ and Fortran (continued)

C and C++ data types	Fortran data types
unsigned long long int	LOGICAL*8
float	REAL REAL*4
double	REAL*8 DOUBLE PRECISION
long double	REAL*8 DOUBLE PRECISION
float _Complex	COMPLEX*8 or COMPLEX(4)
double _Complex	COMPLEX*16 or COMPLEX(8)
long double _Complex	COMPLEX*16 or COMPLEX(8)
structure or union	derived type
enumeration	INTEGER*4
char[n]	CHARACTER*n
array pointer to type, or type []	Dimensioned variable (transposed)
pointer to function	Functional parameter
structure (with **-qalign=packed**)	Sequence derived type

Related information

- **-qldbl128** in *XL C/C++ Compiler Reference*
- **-qalign** in *XL C/C++ Compiler Reference*

Character and aggregate data

Most numeric data types have counterparts across C/C++ and Fortran. However, character and aggregate data types require special treatment:

- C character strings are delimited by a '**\0**' character. In Fortran, all character variables and expressions have a length that is determined at compile time. Whenever Fortran passes a string argument to another routine, it appends a hidden argument that provides the length of the string argument. This length argument must be explicitly declared in C. The C code should not assume a null terminator; the supplied or declared length should always be used.
- C stores array elements in row-major order (array elements in the same row occupy adjacent memory locations). Fortran stores array elements in ascending storage units in column-major order (array elements in the same column occupy adjacent memory locations). Table 9 shows how a two-dimensional array declared by A[3][2] in C and by A(3,2) in Fortran, is stored:

Table 9. Storage of a two-dimensional array

Storage unit	C and C++ element name	Fortran element name
Lowest	A[0][0]	A(1,1)
	A[0][1]	A(2,1)

Table 9. Storage of a two-dimensional array (continued)

Storage unit	C and C++ element name	Fortran element name
	A[1][0]	A(3,1)
	A[1][1]	A(1,2)
	A[2][0]	A(2,2)
Highest	A[2][1]	A(3,2)

- In general, for a multidimensional array, if you list the elements of the array in the order they are laid out in memory, a row-major array will be such that the rightmost index varies fastest, while a column-major array will be such that the leftmost index varies fastest.

Function calls and parameter passing

Functions must be prototyped identically in both C/C++ and Fortran.

In C, by default, all function arguments are passed by value, and the called function receives a copy of the value passed to it. In Fortran, by default, arguments are passed by reference, and the called function receives the address of the value passed to it. You can use the Fortran %VAL built-in function or the VALUE attribute to pass by value. Refer to the *XL Fortran Language Reference* for more information.

For call-by-reference (as in Fortran), the address of the parameter is passed in a register. When passing parameters by reference, if you write C or C++ functions that call a program written in Fortran, all arguments must be pointers, or scalars with the address operator.

Pointers to functions

A function pointer is a data type whose value is a function address. In Fortran, a dummy argument that appears in an EXTERNAL statement is a function pointer. Function pointers are supported in contexts such as the target of a call statement or an actual argument of such a statement.

Sample program: C/C++ calling Fortran

The following example illustrates how program units written in different languages can be combined to create a single program. It also demonstrates parameter passing between C/C++ and Fortran subroutines with different data types as arguments.

```
#include <stdio.h>
extern double add(int *, double [], int *, double []);

double ar1[4]={1.0, 2.0, 3.0, 4.0};
double ar2[4]={5.0, 6.0, 7.0, 8.0};

main()
{
int x, y;
double z;

x = 3;
y = 3;
```

```
z = add(&x, ar1, &y, ar2); /* Call Fortran add routine */
/* Note: Fortran indexes arrays 1..n */
/* C indexes arrays 0..(n-1) */

printf("The sum of %1.0f and %1.0f is %2.0f \n",
ar1[x-1], ar2[y-1], z);
}
```

The Fortran subroutine is:

```
C Fortran function add.f - for C/C++ interlanguage call example

C Compile separately, then link to C/C++ program

REAL*8 FUNCTION ADD (A, B, C, D)
REAL*8 B,D
INTEGER*4 A,C
DIMENSION B(4), D(4)
ADD = B(A) + D(C)
RETURN
END
```

Chapter 3. Aligning data

XL C/C++ provides many mechanisms for specifying data alignment at the levels of individual variables, members of aggregates, entire aggregates, and entire compilation units. If you are porting applications between different platforms, or between 32-bit and 64-bit modes, you will need to take into account the differences between alignment settings available in the different environments, to prevent possible data corruption and deterioration in performance. In particular, vector types have special alignment requirements which, if not followed, can produce incorrect results. That is, vectors need to be aligned according to a 16 byte boundary. For more information, see the *AltiVec Technology Programming Interface Manual*.

Alignment *modes* allow you to set alignment defaults for all data types for a compilation unit (or subsection of a compilation unit), by specifying a predefined suboption. Alignment *modifiers* allow you to set the alignment for specific variables or data types within a compilation unit, by specifying the exact number of bytes that should be used for the alignment.

"Using alignment modes" discusses the default alignment modes for all data types on the different platforms and addressing models; the suboptions and pragmas you can use to change or override the defaults; and rules for the alignment modes for simple variables, aggregates, and bit fields.

"Using alignment modifiers" on page 12 discusses the different specifiers, pragmas, and attributes you can use in your source code to override the alignment mode currently in effect, for specific variable declarations. It also provides the rules governing the precedence of alignment modes and modifiers during compilation.

Related information

- *AltiVec Technology Programming Interface Manual*, available at http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf
- **-qaltivec** in *XL C/C++ Compiler Reference*

Using alignment modes

Each data type supported by XL C/C++ is aligned along byte boundaries according to platform-specific default alignment *modes*. On Linux, the default alignment mode is **linuxppc**.

You can change the default alignment mode, by using any of the following mechanisms:

Set the alignment mode for all variables in a single file or multiple files during compilation

To use this approach, you specify the **-qalign** compiler option during compilation, with one of the suboptions listed in Table 10 on page 10.

Set the alignment mode for all variables in a section of source code

To use this approach, you specify the **#pragma align** or **#pragma options align** directives in the source files, with one of the suboptions listed in Table 10 on page 10. Each directive changes the alignment mode in effect

for all variables that follow the directive until another directive is encountered, or until the end of the compilation unit.

Each of the valid alignment modes is defined in Table 10, which provides the alignment value, in bytes, for scalar variables, for all data types. Where there are differences between 32-bit and 64-bit modes, these are indicated. Also, where there are differences between the first (scalar) member of an aggregate and subsequent members of the aggregate, these are indicated.

Table 10. Alignment settings (values given in bytes)

		Alignment setting	
Data type	**Storage**	**linuxppc**	**bit_packed**
_Bool (C), bool (C++)	1	1	1
char, signed char, unsigned char	1	1	1
wchar_t (32-bit mode)	2	2	1
wchar_t (64-bit mode)	4	4	1
int, unsigned int	4	4	1
short int, unsigned short int	2	2	1
long int, unsigned long int (32-bit mode)	4	4	1
long int, unsigned long int (64-bit mode)	8	8	1
long long	8	8	1
float	4	4	1
double	8	8	1
long double	8	8	1
long double with **-qldbl128**	16	16	1
pointer (32-bit mode)	4	4	1
pointer (64-bit mode)	8	8	1
vector types	16	16	1

If you generate data with an application on one platform and read the data with an application on another platform, it is recommended that you use the **bit_packed** mode, which results in equivalent data alignment on all platforms.

Note: Vectors in a bit-packed structure may not be correctly aligned unless you take extra action to ensure their alignment.

"Alignment of aggregates" discusses the rules for the alignment of entire aggregates and provide examples of aggregate layouts. "Alignment of bit fields" on page 11 discusses additional rules and considerations for the use and alignment of bit fields, and provides an example of bit-packed alignment.

Related information

- **-qalign** and **#pragma align** in the *XL C/C++ Compiler Reference*

Alignment of aggregates

The data contained in Table 10 apply to scalar variables, and variables which are members of aggregates such as structures, unions, and classes. In addition, the following rules apply to aggregate variables, namely structures, unions or classes, as a whole (in the absence of any modifiers):

- For all alignment modes, the *size* of an aggregate is the smallest multiple of its alignment value that can encompass all of the members of the aggregate.
- C Empty aggregates are assigned a size of 0 bytes.
- C++ Empty aggregates are assigned a size of 1 byte. Note that static data members do not participate in the alignment or size of an aggregate; therefore a structure or class containing only a single static data member has a size of 1 byte.
- For all alignment modes, the *alignment* of an aggregate is equal to the largest alignment value of any of its members. With the exception of packed alignment modes, members whose natural alignment is smaller than that of their aggregate's alignment are padded with empty bytes.
- Aligned aggregates can be nested, and the alignment rules applicable to each nested aggregate are determined by the alignment mode that is in effect when a nested aggregate is declared.

Note: C++ The C++ compiler might generate extra fields for classes that contain base classes or virtual functions. Objects of these types might not conform to the usual mappings for aggregates.

For rules on the alignment of aggregates containing bit fields, see "Alignment of bit fields."

Alignment of bit fields

You can declare a bit field as a `_Bool` (C), `bool` (C++), `char`, `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `long long`, or `unsigned long long` data type. The alignment of a bit field depends on its base type and the compilation mode (32-bit or 64-bit).

C The length of a bit field cannot exceed the length of its base type. In extended mode, you can use the `sizeof` operator on a bit field. The `sizeof` operator on a bit field always returns the size of the base type.

C++ The length of a bit field can exceed the length of its base type, but the remaining bits will be used to pad the field, and will not actually store any value.

However, alignment rules for aggregates containing bit fields are different depending on the alignment mode in effect. These rules are described below.

Rules for Linux PowerPC alignment

- Bit fields are allocated from a bit field container. The size of this container is determined by the declared type of the bit field. For example, a `char` bit field uses an 8-bit container, an `int` bit field uses 32 bits, and so on. The container must be large enough to contain the bit field, as the bit field will not be split across containers.
- Containers are aligned in the aggregate as if they start on a natural boundary for that type of container. Bit fields are not necessarily allocated at the start of the container.
- If a zero-length bit field is the first member of an aggregate, it has no effect on the alignment of the aggregate and is overlapped by the next data member. If a zero-length bit field is a non-first member of the aggregate, it pads to the next alignment boundary determined by its base declared type but does not affect the alignment of the aggregate.

- Unnamed bit fields do not affect the alignment of the aggregate.

Rules for bit-packed alignment

- Bit fields have an alignment of 1 byte, and are packed with no default padding between bit fields.
- A zero-length bit field causes the next member to start at the next byte boundary. If the zero-length bit field is already at a byte boundary, the next member starts at this boundary. A non-bit field member that follows a bit field is aligned on the next byte boundary.

Example of bit-packed alignment

For:

```
#pragma options align=bit_packed
struct {
   int a : 8;
   int b : 10;
   int c : 12;
   int d : 4;
   int e : 3;
   int : 0;
   int f : 1;
   char g;
   } A;

pragma options align=reset
```

The size of `A` is 7 bytes. The alignment of `A` is 1 byte. The layout of `A` is:

Member name	Byte offset	Bit offset
a	0	0
b	1	0
c	2	2
d	3	6
e	4	2
f	5	0
g	6	0

Using alignment modifiers

XL C/C++ also provides alignment *modifiers*, which allow you to exercise even finer-grained control over alignment, at the level of declaration or definition of individual variables. Available modifiers are:

#pragma pack(...)

Valid application:
The entire aggregate (as a whole) immediately following the directive.

Effect: Sets the maximum alignment of the members of the aggregate to which it applies, to a specific number of bytes. Also allows a bit-field to cross a container boundary. Used to reduce the effective alignment of the selected aggregate.

Valid values:
When **-qpack_semantic=ibm** is in effect (the default for XL C/C++), **1**, **2**, **4**, **8**, **16**, **nopack**, **pop**, and empty parentheses. The use of empty parentheses has the same functionality as **nopack**. When **-qpack_semantic=gnu** is in

effect (the default when using gxlc and gxlc++ utilities), **[push,]1**, **[push,]2**, **[push,]4**, **[push,]8**, **[push,]16**, **pop**, and empty parentheses.

__attribute__((aligned(n)))

Valid application:
As a *variable* attribute, it applies to a single aggregate (as a whole), namely a structure, union, or class; or to an individual member of an aggregate.[1] As a *type* attribute, it applies to all aggregates declared of that type. If it is applied to a `typedef` declaration, it applies to all instances of that type.[2]

Effect:
Sets the minimum alignment of the specified variable (or variables), to a specific number of bytes. Typically used to increase the effective alignment of the selected variables.

Valid values:
n must be a positive power of 2, or NIL. NIL can be specified as either `__attribute__((aligned()))` or `__attribute__((aligned))`; this is the same as specifying the maximum system alignment (16 bytes on all UNIX® platforms).

__attribute__((packed))

Valid application:
As a *variable* attribute, it applies to simple variables, or individual members of an aggregate, namely a structure, union or class.[1] As a *type* attribute, it applies to all members of all aggregates declared of that type.

Effect: Sets the maximum alignment of the selected variable, or variables, to which it applies, to the smallest possible alignment value, namely one byte for a variable and one bit for a bit field.

__align(n)

Effect: Sets the minimum alignment of the variable or aggregate to which it applies to a specific number of bytes; also effectively increases the amount of storage occupied by the variable. Used to increase the effective alignment of the selected variables.

Valid application:
Applies to simple static (or global) variables or to aggregates as a whole, rather than to individual members of aggregates, unless these are also aggregates.

Valid values:
n must be a positive power of 2. XL C/C++ also allows you to specify a value greater than the system maximum.

Notes:

1. In a comma-separated list of variables in a declaration, if the modifier is placed at the beginning of the declaration, it applies to all the variables in the declaration. Otherwise, it applies only to the variable immediately preceding it.
2. Depending on the placement of the modifier in the declaration of a `struct`, it can apply to the definition of the type, and hence applies to *all* instances of that type; or it can apply to only a single instance of the type. For details, see ″Type Attributes″ in the *XL C/C++ Language Reference*.

When you use alignment modifiers, the interactions between modifiers and modes, and between multiple modifiers, can become complex. The following sections outline precedence guidelines for alignment modifiers, for the following types of variables:

- simple, or scalar, variables, including members of aggregates (structures, unions or classes) and user-defined types created by `typedef` statements.
- aggregate variables (structures, unions or classes)

Related information

- ″The aligned variable attribute″, ″The packed variable attribute″, ″The aligned type attribute″, ″The packed type attribute″, and ″The __align specifier″ in the *XL C/C++ Language Reference*
- **#pragma pack** and **-qpack_semantic** in the *XL C/C++ Compiler Reference*

Guidelines for determining alignment of scalar variables

The following formulas use a ″top-down″ approach to determining the alignment, given the presence of alignment modifiers, for both *non-embedded* (stand-alone) scalar variables and *embedded* scalars (variables declared as members of an aggregate):

Alignment of variable = maximum(*effective type alignment* , *modified alignment value*)

where *effective type alignment* = maximum(maximum(`aligned` type attribute value, `__align` specifier value) , minimum(*type alignment*, `packed` type attribute value))

and *modified alignment value* = maximum(`aligned` variable attribute value, `packed` variable attribute value)

and where *type alignment* is the alignment mode currently in effect when the variable is declared, or the alignment value applied to a type in a `typedef` statement.

In addition, for embedded variables, which can be modified by the **#pragma pack** directive, the following rule applies:

Alignment of variable = minimum(**#pragma pack** value , maximum(*effective type alignment* , *modified alignment value*))

Note: If a type attribute and a variable attribute of the same kind are both specified in a declaration, the second attribute is ignored.

Guidelines for determining alignment of aggregate variables

The following formulas determine the alignment for aggregate variables, namely structures, unions, and classes:

Alignment of variable = maximum(*effective type alignment* , *modified alignment value*)

where *effective type alignment* = maximum(maximum(`aligned` type attribute value, `__align` specifier value) , minimum(*aggregate type alignment*, `packed` type attribute value))

and *modified alignment value* = maximum (`aligned` variable attribute value , `packed` variable attribute value)

and where *aggregate type alignment* = maximum (alignment of all members)

Note: If a type attribute and a variable attribute of the same kind are both specified in a declaration, the second attribute is ignored.

Chapter 4. Handling floating point operations

The following sections provide reference information, portability considerations, and suggested procedures for using compiler options to manage floating-point operations:

Floating-point formats

XL C/C++ supports the following binary floating-point formats:

- 32-bit single precision, with an approximate range of 10^{-38} to 10^{+38} and precision of about 7 decimal digits
- 64-bit double precision, with an approximate range of 10^{-308} to 10^{+308} and precision of about 16 decimal digits
- 128-bit extended precision, with the same range as double-precision values, but with a precision of about 29 decimal digits

Note that the `long double` type may represent either double-precision or extended-precision values, depending on the setting of the **-qldbl128** compiler option. The default is 128 bits. For compatibility with older compilations, you can use **-qnoldbl128** if you need `long double` to be 64 bits.

Related information

- **-qldbl128** in the *XL C/C++ Compiler Reference*

Handling multiply-add operations

By default, the compiler generates a single non-IEEE 754 compatible multiply-add instruction for binary floating-point expressions such as *a+b*c*, partly because one instruction is faster than two. Because no rounding occurs between the multiply and add operations, this may also produce a more precise result. However, the increased precision might lead to different results from those obtained in other environments, and may cause *x*y-x*y* to produce a nonzero result. To avoid these issues, you can suppress the generation of multiply-add instructions by using the **-qfloat=nomaf** option.

Related information

- **-qfloat** in the *XL C/C++ Compiler Reference*

Compiling for strict IEEE conformance

By default, XL C/C++ follows most, but not all of the rules in the IEEE standard. If you compile with the **-qnostrict** option, which is enabled by default at optimization level **-O3** or higher, some IEEE floating-point rules are violated in ways that can improve performance but might affect program correctness. To avoid this issue, and to compile for strict compliance with the IEEE standard, do the following:

- Use the **-qfloat=nomaf** compiler option.
- If the program changes the rounding mode at runtime, use the **-qfloat=rrm** option.
- If the data or program code contains signaling NaN values (NaNS), use the **-qfloat=nans** option. (A signaling NaN is different from a quiet NaN; you must explicitly code it into the program or data or create it by using the **-qinitauto** compiler option.)
- If you compile with **-O3**, **-O4**, or **-O5**, include the option **-qstrict** after it.

Related information

- "Advanced optimization" on page 36
- **-qfloat** in the *XL C/C++ Compiler Reference*
- **-qinitauto** in the *XL C/C++ Compiler Reference*
- **-qstrict** in the *XL C/C++ Compiler Reference*
- **-qinitauto** in the *XL C/C++ Compiler Reference*

Handling floating-point constant folding and rounding

By default, the compiler replaces most operations involving constant operands with their result at compile time. This process is known as constant folding. Additional folding opportunities may occur with optimization or with the **-qnostrict** option. The result of a floating-point operation folded at compile-time normally produces the same result as that obtained at execution time, except in the following cases:

- The compile-time rounding mode is different from the execution-time rounding mode. By default, both are round-to-nearest; however, if your program changes the execution-time rounding mode, to avoid differing results, do either of the following:
  - Change the compile-time rounding mode to match the execution-time mode, by compiling with the appropriate **-y** option. For more information, and an example, see "Matching compile-time and runtime rounding modes" on page 19.
  - Suppress folding, by compiling with the **-qfloat=nofold** option.
- Expressions like *a+b*c* are partially or fully evaluated at compile-time. The results might be different from those produced at execution time, because *b*c* might be rounded before being added to *a*, while the runtime multiply-add instruction does not use any intermediate rounding. To avoid differing results, do either of the following:
  - Suppress the use of multiply-add instructions, by compiling with the **-qfloat=nomaf** option.
  - Suppress folding, by compiling with the **-qfloat=nofold** option.

- An operation produces an infinite or NaN result. Compile-time folding prevents execution-time detection of an exception, even if you compile with the **-qflttrap** option. To avoid missing these exceptions, suppress folding with the **-qfloat=nofold** option.

Related information

- "Handling floating-point exceptions" on page 20
- **-qfloat** and **-qstrict** in the *XL C/C++ Compiler Reference*

Matching compile-time and runtime rounding modes

The default rounding mode used at compile-time and runtime is round-to-nearest, ties even. If your program changes the rounding mode at runtime, the results of a floating-point calculation might be slightly different from those that are obtained at compile-time. The following example illustrates this:

```
#include <float.h>
#include <fenv.h>
#include <stdio.h>

int main ( )
{
 volatile double one = 1.f, three = 3.f;  /* volatiles are not folded */
 double one_third;

 one_third = 1. / 3.;  /* folded */
 printf ("1/3 with compile-time rounding = %.17f\n", one_third);

 fesetround (FE_TOWARDZERO);
 one_third = one / three;  /* not folded */
  printf ("1/3 with execution-time rounding to zero = %.17f\n", one_third);

 fesetround (FE_TONEAREST);
 one_third = one / three;  /* not folded */
  printf ("1/3 with execution-time rounding to nearest = %.17f\n", one_third);

 fesetround (FE_UPWARD);
 one_third = one / three;  /* not folded */
  printf ("1/3 with execution-time rounding to +infinity = %.17f\n", one_third);

 fesetround (FE_DOWNWARD);
 one_third = one / three;  /* not folded */
  printf ("1/3 with execution-time rounding to -infinity = %.17f\n", one_third);

 return 0;
 }
```

When compiled with the default options, this code produces the following results:

```
1/3 with compile-time rounding = 0.33333333333333331
1/3 with execution-time rounding to zero = 0.33333333333333331
1/3 with execution-time rounding to nearest   = 0.33333333333333331
1/3 with execution-time rounding to +infinity = 0.33333333333333337
1/3 with execution-time rounding to -infinity = 0.33333333333333331
```

Because the fourth computation changes the rounding mode to round-to-infinity, the results are slightly different from the first computation, which is performed at compile-time, using round-to-nearest. If you do not use the **-qfloat=nofold** option to suppress all compile-time folding of floating-point computations, it is recommended that you use the **-y** compiler option with the appropriate suboption to match compile-time and runtime rounding modes. In the previous example, compiling with **-yp** (round-to-infinity) produces the following result for the first computation:

```
1/3 with compile-time rounding = 0.33333333333333337
```

In general, if the rounding mode is changed to +infinity or -infinity, it is recommended that you also use the **-qfloat=rrm** option.

Related information

- **-qfloat** and **-y** in the *XL C/C++ Compiler Reference*

Handling floating-point exceptions

By default, invalid operations such as division by zero, division by infinity, overflow, and underflow are ignored at runtime. However, you can use the **-qflttrap** option to detect these types of exceptions. In addition, you can add suitable support code to your program to allow program execution to continue after an exception occurs, and to modify the results of operations causing exceptions.

Because, however, floating-point computations involving constants are usually folded at compile-time, the potential exceptions that would be produced at runtime will not occur. To ensure that the **-qflttrap** option traps all runtime floating-point exceptions, consider using the **-qfloat=nofold** option to suppress all compile-time folding.

Related information

- **-qfloat** and **-qflttrap** in the *XL C/C++ Compiler Reference*

Chapter 5. Using C++ templates

In C++, you can use a template to declare a set of related:

- Classes (including structures)
- Functions
- Static data members of template classes

Within an application, you can instantiate the same template multiple times with the same arguments or with different arguments. If you use the same arguments, the repeated instantiations are redundant. These redundant instantiations increase compilation time, increase the size of the executable, and deliver no benefit.

There are four basic approaches to the problem of redundant instantiations:

Code for unique instantiations
Organize your source code so that the object files contain only one instance of each required instantiation and no unused instantiations. This is the least usable approach, because you must know where each template is defined and where each template instantiation is required.

Instantiate at every occurrence
Use the **-qnotempinc** and **-qnotemplateregistry** compiler options (these are the default settings). The compiler generates code for every instantiation that it encounters. With this approach, you accept the disadvantages of redundant instantiations.

Have the compiler store instantiations in a template include directory
Use the **-qtempinc** compiler option. If the template definition and implementation files have the required structure, each template instantiation is stored in a template include directory. If the compiler is asked to instantiate the same template again with the same arguments, it uses the stored version instead. This approach is described in "Using the -qtempinc compiler option."

Have the compiler store instantiation information in a registry
Use the **-qtemplateregistry** compiler option. Information about each template instantiation is stored in a template registry. If the compiler is asked to instantiate the same template again with the same arguments, it points to the instantiation in the first object file instead. The **-qtemplateregistry** compiler option provides the benefits of the **-qtempinc** compiler option but does not require a specific structure for the template definition and implementation files. This approach is described in "Using the -qtemplateregistry compiler option" on page 24.

Note: The **-qtempinc** and **-qtemplateregistry** compiler options are mutually exclusive.

Related information

- -qtmplinst

Using the -qtempinc compiler option

To use **-qtempinc**, you must structure your application as follows:

1. Declare your class templates and function templates in template header files, with a `.h` extension.
2. For each template declaration file, create a template implementation file. This file must have the same file name as the template declaration file and an extension of `.c` or `.t`, or the name must be specified in a **#pragma implementation** directive. For a class template, the implementation file defines the member functions and static data members. For a function template, the implementation file defines the function.
3. In your source program, specify an `#include` directive for each template declaration file.
4. Optionally, to ensure that your code is applicable for both **-qtempinc** and **-qnotempinc** compilations, in each template declaration file, conditionally include the corresponding template implementation file if the __TEMPINC__ macro is *not* defined. (This macro is automatically defined when you use the **-qtempinc** compilation option.)

   This produces the following results:

   - Whenever you compile with **-qnotempinc**, the template implementation file is included.
   - Whenever you compile with **-qtempinc**, the compiler does not include the template implementation file. Instead, the compiler looks for a file with the same name as the template implementation file and extension `.c` the first time it needs a particular instantiation. If the compiler subsequently needs the same instantiation, it uses the copy stored in the template include directory.

Related information

- **-qtempinc** and **#pragma implementation** in the *XL C/C++ Compiler Reference*

Example of -qtempinc

This example includes the following source files:

- A template declaration file: `stack.h`.
- The corresponding template implementation file: `stack.c`.
- A function prototype: `stackops.h` (not a function template).
- The corresponding function implementation file: `stackops.cpp`.
- The main program source file: `stackadd.cpp`.

In this example:

1. Both source files include the template declaration file `stack.h`.
2. Both source files include the function prototype `stackops.h`.
3. The template declaration file conditionally includes the template implementation file `stack.c` if the program is compiled with **-qnotempinc**.

Template declaration file: stack.h

This header file defines the class template for the class `Stack`.

```
#ifndef STACK_H
#define STACK_H

template <class Item, int size> class Stack {
  public:
      void push(Item item);  // Push operator
      Item pop();            // Pop operator
      int isEmpty(){
          return (top==0);   // Returns true if empty, otherwise false
```

```
      }
      Stack() { top = 0; }   // Constructor defined inline
  private:
      Item stack[size];      // The stack of items
      int   top;             // Index to top of stack
};

#ifndef __TEMPINC__          //  3
#include "stack.c"           //  3
#endif                       //  3
#endif
```

Template implementation file: stack.c

This file provides the implementation of the class template for the class `Stack`.

```
template <class Item, int size>
    void Stack<Item,size>::push(Item item) {
    if (top >= size) throw size;
    stack[top++] = item;
      }
template <class Item, int size>
    Item Stack<Item,size>::pop() {
    if (top <= 0) throw size;
    Item item = stack[--top];
    return(item);
      }
```

Function declaration file: stackops.h

This header file contains the prototype for the `add` function, which is used in both `stackadd.cpp` and `stackops.cpp`.

```
void add(Stack<int, 50>& s);
```

Function implementation file: stackops.cpp

This file provides the implementation of the `add` function, which is called from the main program.

```
#include "stack.h"           //  1
#include "stackops.h"        //  2

void add(Stack<int, 50>& s) {
  int tot = s.pop() + s.pop();
  s.push(tot);
  return;
    }
```

Main program file: stackadd.cpp

This file creates a `Stack` object.

```
#include <iostream.h>
#include "stack.h"           //  1
#include "stackops.h"        //  2

main() {
  Stack<int, 50> s;          // create a stack of ints
  int left=10, right=20;
  int sum;

  s.push(left);              // push 10 on the stack
  s.push(right);             // push 20 on the stack
  add(s);                    // pop the 2 numbers off the stack
                                 // and push the sum onto the stack
  sum = s.pop();             // pop the sum off the stack
```

```
cout << "The sum of: " << left << " and: " << right << " is: " << sum << endl;

return(0);
  }
```

Regenerating the template instantiation file

The compiler builds a template instantiation file in the TEMPINC directory corresponding to each template implementation file. With each compilation, the compiler can add information to the file but it never removes information from the file.

As you develop your program, you might remove template function references or reorganize your program so that the template instantiation files become obsolete. You can periodically delete the TEMPINC destination and recompile your program.

Using -qtempinc with shared libraries

In a traditional application development environment, different applications can share both source files and compiled files. When you use templates, applications can share source files but cannot share compiled files.

If you use **-qtempinc**:

- Each application must have its own TEMPINC destination.
- You must compile all of the source files for the application, even if some of the files have already been compiled for another application.

Using the -qtemplateregistry compiler option

Unlike **-qtempinc**, the **-qtemplateregistry** compiler option does not impose specific requirements on the organization of your source code. Any program that compiles successfully with **-qnotempinc** will compile with **-qtemplateregistry**.

The template registry uses a ″first-come first-served″ algorithm:

- When a program references a new instantiation for the first time, it is instantiated in the compilation unit in which it occurs.
- When another compilation unit references the same instantiation, it is not instantiated. Thus, only one copy is generated for the entire program.

The instantiation information is stored in a template registry file. You must use the same template registry file for the entire program. Two programs cannot share a template registry file.

The default file name for the template registry file is `templateregistry`, but you can specify any other valid file name to override this default. When cleaning your program build environment before starting a fresh or scratch build, you must delete the registry file along with the old object files.

Related information

- **-qtemplateregistry** and **-qtemplaterecompile** in the *XL C/C++ Compiler Reference*

Recompiling related compilation units

If two compilation units, A and B, reference the same instantiation, the **-qtemplateregistry** compiler option has the following effect:

- If you compile A first, the object file for A contains the code for the instantiation.

- When you later compile B, the object file for B does not contain the code for the instantiation because object A already does.
- If you later change A so that it no longer references this instantiation, the reference in object B would produce an unresolved symbol error. When you recompile A, the compiler detects this problem and handles it as follows:
  - If the **-qtemplaterecompile** compiler option is in effect, the compiler automatically recompiles B during the link step, using the same compiler options that were specified for A. (Note, however, that if you use separate compilation and linkage steps, you need to include the compilation options in the link step to ensure the correct compilation of B.)
  - If the **-qnotemplaterecompile** compiler option is in effect, the compiler issues a warning and you must manually recompile B.

Switching from -qtempinc to -qtemplateregistry

Because the **-qtemplateregistry** compiler option does not impose any restrictions on the file structure of your application, it has less administrative overhead than **-qtempinc**. You can make the switch as follows:

- If your application compiles successfully with both **-qtempinc** and **-qnotempinc**, you do not need to make any changes.
- If your application compiles successfully with **-qtempinc** but not with **-qnotempinc**, you must change it so that it will compile successfully with **-qnotempinc**. In each template definition file, conditionally include the corresponding template implementation file if the __TEMPINC__ macro is not defined. This is illustrated in "Example of -qtempinc" on page 22.

Chapter 6. Constructing a library

You can include static and shared libraries in your C and C++ applications.

"Compiling and linking a library" describes how to compile your source files into object files for inclusion in a library, how to link a library into the main program, and how to link one library into another.

"Initializing static objects in libraries (C++)" on page 28 describes how to use priorities to control the order of initialization of objects across multiple files in a C++ application.

Compiling and linking a library

Compiling a static library

To compile a static library:

1. Compile each source file into an object file, with no linking. For example:

   ```
   xlc -c bar.c example.c
   ```

2. Use the **ar** command to add the generated object files to an archive library file. For example:

   ```
   ar -rv libfoo.a bar.o example.o
   ```

Compiling a shared library

To compile a shared library:

1. Compile your source files into an object file, with no linking. Note that in the case of compiling a shared library, the **-qpic** compiler option is also used. For example:

   ```
   xlc -qpic -c foo.c
   ```

2. Use the **-qmkshrobj** compiler option to create a shared object from the generated object files. For example:

   ```
   xlc -qmkshrobj -o libfoo.so foo.o
   ```

Related information

- **-qmkshrobj** in the *XL C/C++ Compiler Reference*

Linking a library to an application

You can use the same command string to link a static or shared library to your main program. For example:

```
xlc -o myprogram main.c -Ldirectory [-Rdirectory] -lfoo
```

where *directory* is the path to the directory containing the library.

By using the **-l** option, you instruct the linker to search in the directory specified via the **-L** option (and, for a shared library, the **-R** option) for `libfoo.so`; if it is not found, the linker searches for `libfoo.a`. For additional linkage options, including options that modify the default behavior, see the operating system **ld** documentation.

Linking a shared library to another shared library

Just as you link modules into an application, you can create dependencies between shared libraries by linking them together. For example:

```
xlc -qmkshrobj -o mylib.so myfile.o -Ldirectory -Rdirectory -lfoo
```

Related information

- **-qmkshrobj**, **-l**, **-R** and **-L** in the *XL C/C++ Compiler Reference*

Initializing static objects in libraries (C++)

The C++ language definition specifies that, before the `main` function in a C++ program is executed, all objects with constructors, from all the files included in the program must be properly constructed. Although the language definition specifies the order of initialization for these objects *within* a file (which follows the order in which they are declared), it does not specify the order of initialization for these objects *across* files and libraries. You might want to specify the initialization order of static objects declared in various files and libraries in your program.

To specify an initialization order for objects, you assign relative *priority* numbers to objects. The mechanisms by which you can specify priorities for entire files or objects within files are discussed in "Assigning priorities to objects." The mechanisms by which you can control the initialization order of objects across modules are discussed in "Order of object initialization across libraries" on page 30.

Assigning priorities to objects

You can assign a priority number to objects and files within a single library, and the objects will be initialized at runtime according to the order of priority. However, because of the differences in the way modules are loaded and objects initialized on the different platforms, the levels at which you can assign priorities vary among the different platforms, as follows:

Set the priority level for an entire file
To use this approach, you specify the **-qpriority** compiler option during compilation. By default, all objects within a single file are assigned the same priority level, and are initialized in the order in which they are declared, and terminated in reverse declaration order.

Set the priority level for objects within a file
To use this approach, you include **#pragma priority** directives in the source files. Each **#pragma priority** directive sets the priority level for all objects that follow it, until another pragma directive is specified. Within a file, the first **#pragma priority** directive must have a higher priority number than the number specified in the **-qpriority** option (if it is used), and subsequent **#pragma priority** directives must have increasing numbers. While the relative priority of objects *within* a single file will remain the order in which they are declared, the pragma directives will affect the order in which objects are initialized *across* files. The objects are initialized according to their priority, and terminated in reverse priority order.

Set the priority level for individual objects
To use this approach, you use `init_priority` variable attributes in the source files. The `init_priority` attribute takes precedence over **#pragma priority** directives, and can be applied to objects in any declaration order. On Linux, the objects are initialized according to their priority and terminated in reverse priority *across* compilation units.

Related information

- "The init_priority variable attribute" in the *XL C/C++ Language Reference*

Using priority numbers

Priority numbers can range from 101 to 65535. The smallest priority number that you can specify, 101, is initialized first. The largest priority number, 65535, is initialized last. If you do not specify a priority level, the default priority is 65535.

The examples below show how to specify the priority of objects within a single file, and across two files. "Order of object initialization across libraries" on page 30 provides detailed information on the order of initialization of objects on the Linux platform.

Example of object initialization within a file

The following example shows how to specify the priority for several objects within a source file.

```
...
#pragma priority(2000) //Following objects constructed with priority 2000
...

static Base a ;

House b ;
...
#pragma priority(3000) //Following objects constructed with priority 3000
...

Barn c ;
...
#pragma priority(2500) // Error - priority number must be larger
                       // than preceding number (3000)
...
#pragma priority(4000) //Following objects constructed with priority 4000
...

Garage d ;
...
```

Example of object initialization across multiple files

The following example describes the initialization order for objects in two files, `farm.C` and `zoo.C`. Both files are contained in the same shared module, and use the **-qpriority** compiler option and **#pragma priority** directives.

farm.C -qpriority=1000

```
...
Dog a ;
Dog b ;
...
#pragma priority(6000)
...
Cat c ;
Cow d ;
...
#pragma priority(7000)
Mouse e ;
...
```

zoo.C -qpriority=2000

```
...

Bear m ;
...
#pragma priority(5000)
...
Zebra n ;
Snake s ;
...
#pragma priority(8000)
Frog f ;
...
```

At runtime, the objects in these files are initialized in the following order:

Sequence	Object	Priority value	Comment
1	Dog a	1000	Takes option priority (1000).
2	Dog b	1000	Follows with the same priority.
3	Bear m	2000	Takes option priority (2000).
4	Zebra n	5000	Takes pragma priority (5000).
5	Snake s	5000	Follows with same priority.
6	Cat c	6000	Next priority number.
7	Cow d	6000	Follows with same priority.
8	Mouse e	7000	Next priority number.
9	Frog f	8000	Next priority number (initialized last).

Order of object initialization across libraries

Each static library and shared library is loaded and initialized at runtime in *reverse* link order, once all of its dependencies have been loaded and initialized. Link order is the order in which each library was listed on the command line during linking into the main application. For example, if library A calls library B, library B is loaded before library A.

As each module is loaded, objects are initialized in order of priority, according to the rules outlined in "Assigning priorities to objects" on page 28. If objects do not have priorities assigned, or have the same priorities, object files are initialized in reverse link order — where link order is the order in which the files were given on the command line during linking into the library — and the objects within the files are initialized according to their declaration order. Objects are terminated in reverse order of their construction.

Example of object initialization across libraries

In this example, the following modules are used:

- `main.out`, the executable containing the main function
- `libS1` and `libS2`, two shared libraries
- `libS3` and `libS4`, two shared libraries that are dependencies of `libS1`
- `libS5` and `libS6`, two shared libraries that are dependencies of `libS2`

The source files are compiled into object files with the following command strings:

```
xlC -qpriority=101 -c fileA.C -o fileA.o
xlC -qpriority=150 -c fileB.C -o fileB.o
xlC -c fileC.C -o fileC.o
xlC -c fileD.C -o fileD.o
xlC -c fileE.C -o fileE.o
xlC -c fileF.C -o fileF.o
xlC -qpriority=300 -c fileG.C -o fileG.o
xlC -qpriority=200 -c fileH.C -o fileH.o
xlC -qpriority=500 -c fileI.C -o fileI.o
xlC -c fileJ.C -o fileJ.o
xlC -c fileK.C -o fileK.o
xlC -qpriority=600 -c fileL.C -o fileL.o
```

The dependent libraries are created with the following command strings:

```
xlC -qmkshrobj -o libS3.so fileE.o fileF.o
xlC -qmkshrobj -o libS4.so fileG.o fileH.o
xlC -qmkshrobj -o libS5.so fileI.o fileJ.o
xlC -qmkshrobj -o libS6.so fileK.o fileL.o
```

The dependent libraries are linked with their parent libraries using the following command strings:

```
xlC -qmkshrobj -o libS1.so fileA.o fileB.o -L. -R. -lS3 -lS4
xlC -qmkshrobj -o libS2.so fileC.o fileD.o -L. -R. -lS5 -lS6
```

The parent libraries are linked with the main program with the following command string:

```
xlC main.C -o main.out -L. -R. -lS1 -lS2
```

The following diagram shows the initialization order of the shared libraries.

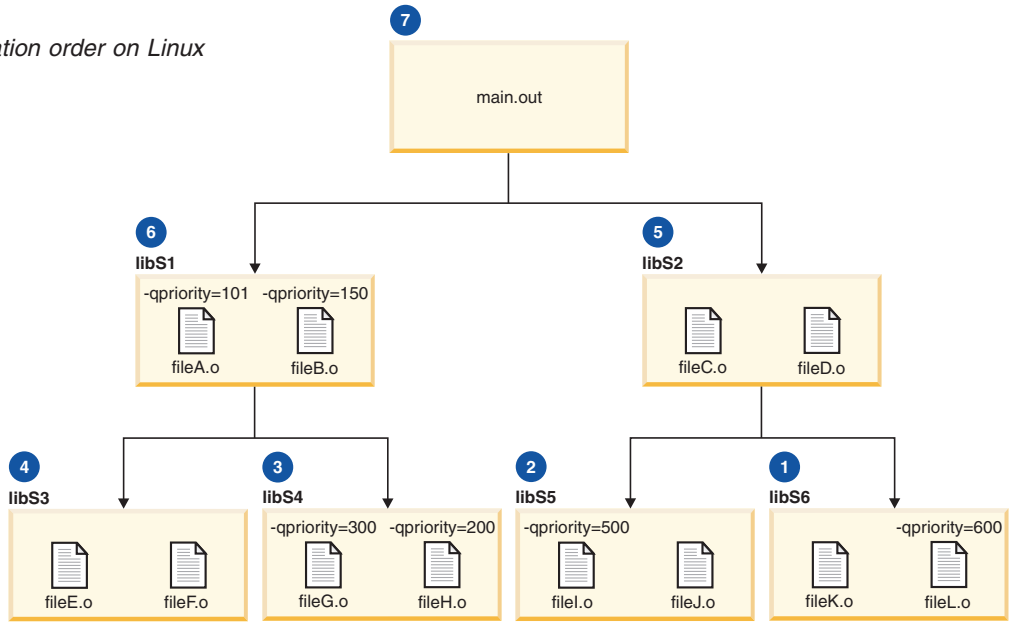


Figure 1. Object initialization order on Linux

Objects are initialized as follows:

Sequence	Object	Priority value	Comment
1	libS6	n/a	`libS2` was entered last on the command line when linked with `main`, and so is initialized before `libS1`. However, `libS5` and `libS6` are dependencies of `libS2`, so they are initialized first. Since it was entered last on the command line when linked to create `libS2`, `libS6` is initialized first. The objects in this library are initialized according to their priority.
2	fileL	600	The objects in `fileL` are initialized next (lowest priority number in this module).
3	fileK	65535	The objects in `fileK` are initialized next (next priority number in this module (default priority of 65535)).

Sequence	Object	Priority value	Comment
4	libS5	n/a	`libS5` was entered before `libS6` on the command line when linked with `libS2`, so it is initialized next. The objects in this library are initialized according to their priority.
5	fileI	500	The objects in `fileI` are initialized next (lowest priority number in this module).
6	fileJ	65535	The objects in `fileJ` are initialized next (next priority number in this module (default priority of 65535)).
7	libS4	n/a	`libS4` is a dependency of `libS1` and was entered last on the command line when linked to create `libS1`, so it is initialized next. The objects in this library are initialized according to their priority.
8	fileH	200	The objects in `fileH` are initialized next (lowest priority number in this module).
9	fileG	300	The objects in `fileG` are initialized next (next priority number in this module).
10	libS3	n/a	`libS3` is a dependency of `libS1` and was entered first on the command line during the linking with `libS1`, so it is initialized next. The objects in this library are initialized according to their priority.
11	fileF	65535	Both `fileF` and `fileE` are assigned a default priority of 65535. However, because `fileF` was listed last on the command line when the object files were linked into `libS3`, `fileF` is initialized first.
12	fileE	65535	Initialized next.
13	libS2	n/a	`libS2` is initialized next. The objects in this library are initialized according to their priority.
14	fileD	65535	Both `fileD` and `fileC` are assigned a default priority of 65535. However, because `fileD` was listed last on the command line when the object files were linked into `libS2`, `fileD` is initialized first.
15	fileC	65535	Initialized next.
16	libS1		`libS1` is initialized next. The objects in this library are initialized according to their priority.
17	fileA	101	The objects in `fileA` are initialized next (lowest priority number in this module).
18	fileB	150	The objects in `fileB` are initialized next (next priority number in this module).
19	main.out	n/a	Initialized last. The objects in `main.out` are initialized according to their priority.

Chapter 7. Optimizing your applications

The XL compilers enable development of high performance 32-bit and 64-bit applications for the Linux operating system by offering a comprehensive set of performance enhancing techniques that exploit the multilayered PowerPC® architecture. These performance advantages depend on good programming techniques, thorough testing and debugging, followed by optimization, and tuning.

Distinguishing between optimization and tuning

You can use optimization and tuning separately or in combination to increase the performance of your application. Understanding the difference between them is the first step in understanding how the different levels, settings and techniques can increase performance.

Optimization

Optimization is a compiler driven process that searches for opportunities to restructure your source code and give your application better overall performance at runtime, without significantly impacting development time. The XL compiler optimization suite, which you control using compiler options and directives, performs best on well-written source code that has already been through a thorough debugging and testing process. These optimization transformations can:

- Reduce the number of instructions your application executes to perform critical operations.
- Restructure your object code to make optimal use of the PowerPC architecture.
- Improve memory subsystem usage.
- Exploit the ability of the architecture to handle large amounts of shared memory parallelization.

Consider that although not all optimizations benefit all applications, even basic optimization techniques can result in a performance benefit. Consult the Steps in the optimization process for an overview of the common sequence of steps you can use to increase the performance of your application.

Tuning

Where optimization applies increasingly aggressive transformations designed to improve the performance of any application in any supported environment, tuning offers you opportunities to adjust characteristics of your application to improve performance, or to target specific execution environments. Even at low optimization levels, tuning for your application and target architecture can have a positive impact on performance. With proper tuning the compiler can:

- Select more efficient machine instructions.
- Generate instruction sequences that are more relevant to your application.

For instructions, see Tuning for your system architecture.

Steps in the optimization process

As you begin the optimization process, consider that not all optimization techniques suit all applications. Trade-offs sometimes occur between an increase in compile time, a reduction in debugging capability, and the improvements that optimization can provide. Learning about, and experimenting with different optimization techniques can help you strike the right balance for your XL compiler applications while achieving the best possible performance. Also, though it is unnecessary to hand-optimize your code, compiler-friendly programming can be extremely beneficial to the optimization process. Unusual constructs can obscure the characteristics of your application and make performance optimization difficult. Use the steps in this section as a guide for optimizing your application.

1. The Basic optimization step begins your optimization processes at levels 0 and 2.
2. The Advanced optimization step exposes your application to more intense optimizations at levels 3 through 5.
3. The Using high-order loop analysis and transformations step can help you limit loop execution time.
4. The Using interprocedural analysis, step can optimize your entire application at once.
5. The Using profile-directed feedback step focuses optimizations on specific characteristics of your application.
6. The Debugging high-performance code step can help you identify issues and problems that can occur with optimized code.

Basic optimization

The XL compiler supports several levels of optimization, with each option level building on the levels below through increasingly aggressive transformations, and consequently using more machine resources. Ensure that your application compiles and executes properly at low optimization levels before trying more aggressive optimizations. This section discusses two optimizations levels, listed with complementary options in the *Basic optimizations* table. The table also includes a column for compiler options that can have a performance benefit at that optimization level for some applications.

Table 11. Basic optimizations

Optimization level	Additional options implied by default	Complementary options	Other options with possible benefits
-O0	None	**-qarch**	**-g**
-O2	**-qmaxmem=**8192	**-qarch** **-qtune**	**-qmaxmem=**-1 **-qhot=level=0**

Optimizing at level 0

Benefits at level 0

- Minimal performance improvement, with minimal impact on machine resources.
- Exposes some source code problems, helping in the debugging process.

Begin your optimization process at **-O0** which the compiler already specifies by default. For SMP programs, the closest equivalent to **-O0** is **-qsmp=noopt**. This level performs basic analytical optimization by removing obviously redundant code, and can result in better compile time, while ensuring your code is algorithmically correct so you can move forward to more complex optimizations. **-O0** also includes constant folding. The option **-qfloat=nofold** can be used to suppress folding floating-point operations. Optimizing at this level accurately preserves all debug information and can expose problems in existing code, such as uninitialized variables and bad casting.

Additionally, specifying **-qarch** at this level targets your application for a particular machine and can significantly improve performance by ensuring your application takes advantage of all applicable architectural benefits.

For more information on tuning, consult Tuning for your system architecture.

Optimizing at level 2

Benefits at level 2

- Eliminates redundant code
- Basic loop optimization
- Can structure code to take advantage of **-qarch** and **-qtune** settings

After successfully compiling, executing, and debugging your application using **-O0**, recompiling at **-O2** opens your application to a set of comprehensive low-level transformations that apply to subprogram or compilation unit scopes and can include some inlining. Optimizations at **-O2** are a relative balance between increasing performance while limiting the impact on compilation time and system resources. You can increase the memory available to some of the optimizations in the **-O2** portfolio by providing a larger value for the **-qmaxmem** option. Specifying **-qmaxmem=-1** allows the optimizer to use memory as needed without checking for limits but does not change the transformations the optimizer applies to your application at **-O2**.

In C, compile with **-qlibansi** unless your application defines functions with names identical to those of library functions. If you encounter problems with **-O2**, consider using **-qalias=noansi** rather than turning off optimization.

Also, ensure that pointers in your C code follow these type restrictions:

- Generic pointers can be `char*` or `void*`
- Mark all shared variables and pointers to shared variables `volatile`

Starting to tune at O2

Choosing the right hardware architecture target or family of targets becomes even more important at **-O2** and higher. Targeting the proper hardware allows the optimizer to make the best use of the hardware facilities available. If you choose a family of hardware targets, the **-qtune** option can direct the compiler to emit code consistent with the architecture choice, but will execute optimally on the chosen tuning hardware target. This allows you to compile for a general set of targets but have the code run best on a particular target.

See the Tuning for your system architecture section for details on the **-qarch** and **-qtune** options.

The **-O2** option can perform a number of additional optimizations, including:

- Common subexpression elimination: Eliminates redundant instructions.
- Constant propagation: Evaluates constant expressions at compile-time.
- Dead code elimination: Eliminates instructions that a particular control flow does not reach, or that generate an unused result.
- Dead store elimination: Eliminates unnecessary variable assignments.
- Graph coloring register allocation: Globally assigns user variables to registers.
- Value numbering: Simplifies algebraic expressions, by eliminating redundant computations.
- Instruction scheduling for the target machine.
- Loop unrolling and software pipelining.
- Moves invariant code out of loops.
- Simplifies control flow.
- Strength reduction and effective use of addressing modes.

Even with **-O2** optimizations, some useful information about your source code is made available to the debugger if you specify **-g**. Conversely, higher optimization levels can transform code to an extent to which debug information is no longer accurate. Use that information with discretion.

Advanced optimization

After applying basic optimizations and successfully compiling and executing your application, you can apply more powerful optimization tools. Higher optimization levels can have a tremendous impact on performance, but some trade-offs can occur in terms of code size, compilation time, resource requirements and numeric or algorithmic precision. The XL compiler optimization portfolio includes many options for directing advanced optimization, and the transformations your application undergoes are largely under your control. The discussion of each optimization level in Table 12 on page 36 includes information on not only the performance benefits, and the possible trade-offs as well, but information on how you can help guide the optimizer to find the best solutions for your application.

Table 12. Advanced optimizations

Optimization Level	Additional options implied	Complementary options	Options with possible benefits
-O3	**-qnostrict** **-qmaxmem=-1** **-qhot=level=0**	**-qarch** **-qtune**	**-qpdf**
-O4	**-qnostrict** **-qmaxmem=-1** **-qhot** **-qipa** **-qarch=auto** **-qtune=auto** **-qcache=auto**	**-qarch** **-qtune** **-qcache**	**-qpdf** **-qsmp=auto**
-O5	All of **-O4** **-qipa=level=2**	**-qarch** **-qtune** **-qcache**	**-qpdf** **-qsmp=auto**

Optimizing at level 3

Benefits at level 3

- In-depth memory access analysis
- Better loop scheduling
- High-order loop analysis and transformations (**-qhot=level=0**)
- Inlining of small procedures within a compilation unit by default
- Eliminating implicit compile-time memory usage limits
- Widening, which merges adjacent load/stores and other operations
- Pointer aliasing improvements to enhance other optimizations

Specifying **-O3** initiates more intense low-level transformations that remove many of the limitations present at **-O2**. For instance, the optimizer no longer checks for memory limits, by defaulting to **-qmaxmem=-1**. Additionally, optimizations encompass larger program regions and attempt more in-depth analysis. While not all applications contain opportunities for the optimizer to provide a measurable increase in performance, most applications can benefit from this type of analysis.

Potential trade-offs at level 3

With the in-depth analysis of **-O3** comes a trade-off in terms of compilation time and memory resources. Also, since **-O3** implies **-qnostrict**, the optimizer can alter certain floating-point semantics in your application to gain execution speed. This typically involves precision trade-offs as follows:

- Reordering of floating-point computations.
- Reordering or elimination of possible exceptions, such as division by zero or overflow.

You can still gain most of the **-O3** benefits while preserving precise floating-point semantics by specifying **-qstrict**. Compiling with **-qstrict** is necessary if you require the same absolute precision in floating-point computational accuracy as you get with **-O0**, **-O2**, or **-qnoopt** results. The **-qstrict** compiler option also ensures adherence to all IEEE semantics for floating-point operations. If your application is sensitive to floating-point exceptions or the order of evaluation for floating-point arithmetic, compiling with **-qstrict** will help assure accurate results. Without **-qstrict**, the difference in computation for any one source-level operation is very small in comparison to basic optimization. Though a small difference can compound if the operation is in a loop structure where the difference becomes additive, most applications are not sensitive to the changes that can occur in floating-point semantics.

An intermediate step: adding -qhot suboptions at level 3

At **-O3**, the optimization includes minimal **-qhot** loop transformations at **level=0** to increase performance. You can further increase your performance benefit by increasing the level and therefore the aggressiveness of **-qhot**. Try specifying **-qhot** without any suboptions, or **-qhot=level=1**.

For more information on **-qhot**, see Using high-order loop analysis and transformations .

Optimizing at level 4

Benefits at level 4

- Propagation of global and parameter values between compilation units
- Inlining code from one compilation unit to another
- Reorganization or elimination of global data structures
- An increase in the precision of aliasing analysis

Optimizing at **-O4** builds on **-O3** by triggering **-qipa=level=1** which performs interprocedural analysis (IPA), optimizing your entire application as a unit. This option is particularly pertinent to applications that contain a large number of frequently used routines.

To make full use of IPA optimizations, you must specify **-O4** on the compilation and link steps of your application build as interprocedural analysis occurs in stages at both compile and link time.

The IPA process

1. At compilation time optimizations occur on a file-by-file basis, as well as preparation for the link stage. IPA writes analysis information directly into the object files the compiler produces.
2. At the link stage, IPA reads the information from the object files and analyzes the entire application.
3. This analysis guides the optimizer on how to rewrite and restructure your application and apply appropriate **-O3** level optimizations.

The Using interprocedural analysis section contains more information on IPA including details on IPA suboptions.

Beyond **-qipa**, **-O4** enables other optimization options:

- **-qhot**

  Enables more aggressive HOT transformations to optimize loop constructs and array language.

- **-qarch=**auto and **-qtune=**auto

  Optimizes your application to execute on a hardware architecture identical to your build machine. If the architecture of your build machine is incompatible with your application's execution environment, you must specify a different **-qarch** suboption after the **-O4** option. This overrides **-qarch=auto**.

- **-qcache=auto**

  Optimizes your cache configuration for execution on specific hardware architecture. The auto suboption assumes that the cache configuration of your build machine is identical to the configuration of your execution architecture. Specifying a cache configuration can increase program performance, particularly loop operations by blocking them to process only the amount of data that can fit into the data cache.

  If you will be executing your application on a different machine, specify correct cache values.

Potential trade-offs at level 4

In addition to the trade-offs already mentioned for **-O3**, specifying **-qipa** can significantly increase compilation time, especially at the link step.

Optimizing at level 5

Benefits at level 5

- Most aggressive optimizations available
- Makes full use of loop optimizations and IPA

As the highest optimization level, **-O5** includes all **-O4** optimizations and deepens whole program analysis by increasing the **-qipa** level to 2. Compiling with **-O5** also increases how aggressively the optimizer pursues aliasing improvements. Additionally, if your application contains a mix of XL C/C++ and Fortran code that you compile using XL compilers, you can increase performance by compiling and linking your code with the **-O5** option.

Potential trade-offs at level 5

Compiling at **-O5** requires more compilation time and machine resources than any other optimization level, particularly if you include **-O5** on the IPA link step. Compile at **-O5** as the final phase in your optimization process after successfully compiling and executing your application at **-O4**.

Tuning for your system architecture

You can instruct the compiler to generate code for optimal execution on a given microprocessor or architecture family. By selecting appropriate target machine options, you can optimize to suit the broadest possible selection of target processors, a range of processors within a given family of processor architectures, or a specific processor. The following table lists the optimization options that affect individual aspects of the target machine. Using a predefined optimization level sets default values for these individual options.

Table 13. Target machine options

Option	Behavior
-q32	Generates code for a 32-bit (4 byte integer / 4 byte long / 4 byte pointer) addressing model (32-bit execution mode). This is the default setting.
-q64	Generates code for a 64-bit (4 byte integer / 8 byte long / 8 byte pointer) addressing model (64-bit execution mode).
-qarch	Selects a family of processor architectures for which instruction code should be generated. This option restricts the instruction set generated to a subset of that for the PowerPC architecture. The default on all Linux distributions is **-qarch=ppc64grsq**. Using **-O4** or **-O5** sets the default to **-qarch=auto**. See “Getting the most out of target machine options” on page 40 below for more information on this option.
-qipa=clonearch	Allows you to specify multiple specific processor architectures for which instruction sets will be generated. At runtime, the application will detect the specific architecture of the operating environment and select the instruction set specialized for that architecture. The advantage of this option is that it allows you to optimize for several architectures without recompiling your code for each target architecture. See “Using interprocedural analysis ” on page 43 for more information on this option.
-qtune	Biases optimization toward execution on a given microprocessor, without implying anything about the instruction set architecture to use as a target. See “Getting the most out of target machine options” on page 40 below for more information on this option.

Table 13. Target machine options (continued)

Option	Behavior
-qcache	Defines a specific cache or memory geometry. The defaults are determined through the setting of **-qtune**. See "Getting the most out of target machine options" below for more information on this option.

For a complete listing of valid hardware-related suboptions and combinations of suboptions, see "Specifying Compiler Options for Architecture-Specific, 32- or 64-bit Compilation", and "Acceptable -qarch/-qtune combinations" in the *XL C/C++ Compiler Reference*.

Getting the most out of target machine options

Using -qarch options

If your application will run on the same machine on which you are compiling it, you can use the **-qarch=auto** option, which automatically detects the specific architecture of the compiling machine, and generates code to take advantage of instructions available only on that machine (or on a system that supports the equivalent processor architecture). Otherwise, try to specify with **-qarch** the smallest family of machines possible that will be expected to run your code reasonably well, or use the **-qipa=clonearch** option, which will generate instructions for multiple architectures. Note that if you use **-qipa=clonearch**, the **-qarch** value must be in the family of architectures specified by the **clonearch** suboption.

Using -qtune options

If you specify a particular architecture with **-qarch**, **-qtune** will automatically select the suboption that generates instruction sequences with the best performance for that architecture. If you specify a *group* of architectures with **-qarch**, compiling with **-qtune=auto** will generate code that runs on all of the architectures in the specified group, but the instruction sequences will be those with the best performance on the architecture of the compiling machine.

Try to specify with **-qtune** the particular architecture that the compiler should target for best performance but still allow execution of the produced object file on all architectures specified in the **-qarch** option. For information on the valid combinations of **-qarch** and **-qtune**, see "Acceptable -qarch/-qtune combinations" in the *XL C/C++ Compiler Reference*.

If you need to create a single binary that will run on a range of PowerPC hardware, consider using the **-qtune=balanced** option. With this option in effect, optimization decisions made by the compiler are not targeted to a specific version of hardware. Instead, tuning decisions try to include features that are generally helpful across a broad range of hardware and avoid those optimizations that may be harmful on some hardware. Note that you should verify the performance of code compiled with the **-qtune=balanced** option before distributing it.

Using -qcache options

Before using the **-qcache** option, use the **-qlistopt** option to generate a listing of the current settings and verify if they are satisfactory. If you decide to specify your own **-qcache** suboptions, use **-qhot** or **-qsmp** along with it. For the full set of suboptions, option syntax, and guidelines for use, see **-qcache** in the *XL C/C++ Compiler Reference*.

Related information

- "Using the Mathematical Acceleration Subsystem libraries (MASS) " on page 63
- **-qarch**, **-qcache**, **-qtune**, and **-qlistopt** in the *XL C/C++ Compiler Reference*

Using high-order loop analysis and transformations

High-order transformations are optimizations that specifically improve the performance of loops through techniques such as interchange, fusion, and unrolling. The goals of these loop optimizations include:

- Reducing the costs of memory access through the effective use of caches and translation look-aside buffers.
- Overlapping computation and memory access through effective utilization of the data prefetching capabilities provided by the hardware.
- Improving the utilization of microprocessor resources through reordering and balancing the usage of instructions with complementary resource requirements.
- Generating vector instructions.

To enable high-order loop analysis and transformations, you use the **-qhot** option, which implies an optimization level of **-O2**. The following table lists the suboptions available for **-qhot**.

Table 14. **-qhot** *suboptions*

Suboption	Behavior
level=1	This is the default suboption if you specify **-qhot** with no suboptions. This level is also automatically enabled if you compile with **-O4** or **-O5**. This is equivalent to specifying **-qhot=vector** and **-qhot=simd**.
level=0	Instructs the compiler to perform a subset of high-order transformations that enhance performance by improving data locality. This suboption implies **-qhot=novector**, **-qhot=noarraypad** and **-qhot=nosimd**. This level is automatically enabled if you compile with **-O3**.
vector	When specified with **-qnostrict** and **-qignerrno**, or **-O3** or a higher optimization level, instructs the compiler to transform some loops to use the optimized versions of various math functions contained in the MASS libraries, rather than use the system versions. The optimized versions make different trade-offs with respect to accuracy and exception-handling versus performance. This suboption is enabled by default if you specify **-qhot** with no suboptions. Also, specifying **-qhot=vector** with **-O3** implies **-qhot=level=1**.
arraypad	Instructs the compiler to pad any arrays where it infers there might be a benefit and to pad by whatever amount it chooses.
simd	Instructs the compiler to attempt automatic SIMD vectorization; that is, converting certain operations in a loop that apply to successive elements of an array into a call to a VMX instruction. This call calculates several results at one time, which is faster than calculating each result sequentially. This suboption is enabled by default on Linux if you set **-qarch** to a target architecture that supports VMX instructions (and **-qenablevmx** is in effect, which it is by default).

Getting the most out of -qhot

Here are some suggestions for using **-qhot**:

- Try using **-qhot** along with **-O3** for all of your code. It is designed to have a neutral effect when no opportunities for transformation exist.

- If the runtime performance of your code can significantly benefit from automatic inlining and memory locality optimizations, try using **-O4** with **-qhot=level=0** or **-qhot=novector**.
- If you encounter unacceptably long compile times (this can happen with complex loop nests), try **-qhot=level=0**.
- If your code size is unacceptably large, try using **-qcompact** along with **-qhot**.
- If necessary, deactivate **-qhot** selectively, allowing it to improve some of your code.

Related information

- **-qhot**, **-qenablevmx**, and **-qstrict** in *XL C/C++ Compiler Reference*

Using shared-memory parallelism (SMP)

Some IBM pSeries® machines are capable of shared-memory parallel processing. You can compile with **-qsmp** to generate the threaded code needed to exploit this capability. The option implies an optimization level of at least **-O2**.

The following table lists the most commonly used suboptions. Descriptions and syntax of all the suboptions are provided in **-qsmp** in the *XL C/C++ Compiler Reference*. An overview of automatic parallelization, as well as of OpenMP directives is provided in Chapter 11, "Parallelizing your programs," on page 75.

Table 15. Commonly used **-qsmp** *suboptions*

suboption	Behavior
auto	Instructs the compiler to automatically generate parallel code where possible without user assistance. Any SMP programming constructs in the source code, including OpenMP directives, are also recognized. This is the default setting if you do not specify any **-qsmp** suboptions, and it also implies the **opt** suboption.
omp	Instructs the compiler to enforce strict conformance to the OpenMP API for specifying explicit parallelism. Only language constructs that conform to the OpenMP standard are recognized. Note that **-qsmp=omp** is currently incompatible with **-qsmp=auto**.
opt	Instructs the compiler to optimize as well as parallelize. The optimization is equivalent to **-O2 -qhot** in the absence of other optimization options.
noopt	All optimization is turned off. During development, it can be useful to turn off optimization to facilitate debugging.
fine_tuning	Other values for the suboption provide control over thread scheduling, nested parallelism, locking, etc.

Getting the most out of -qsmp

Here are some suggestions for using the **-qsmp** option:

- Before using **-qsmp** with automatic parallelization, test your programs using optimization and **-qhot** in a single-threaded manner.
- If you are compiling an OpenMP program and do not want automatic parallelization, use **-qsmp=omp:noauto** .
- Always use the reentrant compiler invocations (the *_r* invocations) when using **-qsmp**.
- By default, the runtime environment uses all available processors. Do not set the XLSMPOPTS=PARTHDS or OMP_NUM_THREADS environment variables

unless you want to use fewer than the number of available processors. You might want to set the number of executing threads to a small number or to 1 to ease debugging.
- If you are using a dedicated machine or node, consider setting the SPINS and YIELDS environment variables (suboptions of the XLSMPOPTS environment variable) to 0. Doing so prevents the operating system from intervening in the scheduling of threads across synchronization boundaries such as barriers.
- When debugging an OpenMP program, try using **-qsmp=noopt** (without **-O**) to make the debugging information produced by the compiler more precise.

Related information

- "Environment variables for parallel processing" in *XL C/C++ Compiler Reference*
- "Invoking the compiler" in *XL C/C++ Compiler Reference*

Using interprocedural analysis

Interprocedural analysis (IPA) enables the compiler to optimize across different files (whole-program analysis), and can result in significant performance improvements. You can specify interprocedural analysis on the compile step only or on both compile and link steps in "whole program" mode (with the exception of the **clonearch** and **cloneproc** suboptions, which must be specified on the link step). Whole program mode expands the scope of optimization to an entire program unit, which can be an executable or shared object. As IPA can significantly increase compilation time, you should limit using IPA to the final performance tuning stage of development.

You enable IPA by specifying the **-qipa** option. The most commonly used suboptions and their effects are described in the following table. The full set of suboptions and syntax is described in the **-qipa** section of the *XL C/C++ Compiler Reference*.

The steps to use IPA are:

1. Do preliminary performance analysis and tuning before compiling with the **-qipa** option, because the IPA analysis uses a two-pass mechanism that increases compile and link time. You can reduce some compile and link overhead by using the **-qipa=noobject** option.
2. Specify the **-qipa** option on both the compile and the link steps of the entire application, or as much of it as possible. Use suboptions to indicate assumptions to be made about parts of the program *not* compiled with **-qipa**.

Table 16. Commonly used **-qipa** *suboptions*

Suboption	Behavior
level=0	Program partitioning and simple interprocedural optimization, which consists of: • Automatic recognition of standard libraries. • Localization of statically bound variables and procedures. • Partitioning and layout of procedures according to their calling relationships. (Procedures that call each other frequently are located closer together in memory.) • Expansion of scope for some optimizations, notably register allocation.

Table 16. Commonly used **-qipa** *suboptions (continued)*

Suboption	Behavior
level=1	Inlining and global data mapping. Specifically: • Procedure inlining. • Partitioning and layout of static data according to reference affinity. (Data that is frequently referenced together will be located closer together in memory.) This is the default level if you do not specify any suboptions with the **-qipa** option.
level=2	Global alias analysis, specialization, interprocedural data flow: • Whole-program alias analysis. This level includes the disambiguation of pointer dereferences and indirect function calls, and the refinement of information about the side effects of a function call. • Intensive intraprocedural optimizations. This can take the form of value numbering, code propagation and simplification, moving code into conditions or out of loops, and elimination of redundancy. • Interprocedural constant propagation, dead code elimination, pointer analysis, code motion across functions, and interprocedural strength reduction. • Procedure specialization (cloning). • Whole program data reorganization.
inline=*suboptions*	Allows precise control over function inlining.
clonearch=*arch_list*	Allows you to specify multiple architectures for which optimized instructions can be generated. Supported architecture values are **PWR4**, **PWR5**, **PWR6**, and **PPC970**. For every function in your program, the compiler generates a generic version of the instruction set, according to the **-qarch** value in effect, and, if appropriate, *clones* specialized versions of the instruction set for the architectures you specify in this suboption. The compiler inserts code into your application to check for the processor architecture at run time, and selects the version of the generated instructions that is optimized for the runtime environment.
cloneproc=*func_list*	Allows you to specify the exact functions which should be cloned for the specified architectures in the **clonearch** suboption.
fine_tuning	Other values for **-qipa** provide the ability to specify the behavior of library code, tune program partitioning, read commands from a file, etc.

Getting the most from -qipa

It is not necessary to compile everything with **-qipa**, but try to apply it to as much of your program as possible. Here are some suggestions:

- Specify the **-qipa** option on both the compile and link steps of the entire application. Although you can also use **-qipa** with libraries, shared objects, and executable files, be sure to use **-qipa** to compile the main and exported functions.
- When compiling and linking separately, use **-qipa=noobject** on the compile step for faster compilation.
- When specifying optimization options in a makefile, remember to use the compiler driver (**xlc**) to link, and to include all compiler options on the link step.

- As IPA can generate significantly larger object files than traditional compilations, ensure that there is enough space in the /tmp directory (at least 200 MB). You can use the TMPDIR environment variable to specify a directory with sufficient free space.
- Try varying the **level** suboption if link time is too long. Compiling with **–qipa=level=0** can still be very beneficial for little additional link time.
- Use **-qipa=list=long** to generate a report of functions that were inlined. If too few or too many functions are inlined, consider using **–qipa=inline** or **–qipa=noinline**. To control inlining of specific functions, use **-qipa=[no]inline=***function_name*.

Note: While IPA's interprocedural optimizations can significantly improve performance of a program, they can also cause incorrect but previously functioning programs to fail. Here are examples of programming practices that can work by accident without aggressive optimization but are exposed with IPA:

- Relying on the allocation order or location of automatic variables, such as taking the address of an automatic variable and then later comparing it with the address of another local variable to determine the growth direction of a stack. The C language does not guarantee where an automatic variable is allocated, or its position relative to other automatic variables. Do not compile such a function with IPA.
- Accessing a pointer that is either invalid or beyond an array's bounds. Because IPA can reorganize global data structures, a wayward pointer which might have previously modified unused memory might now conflict with user-allocated storage.

Related information

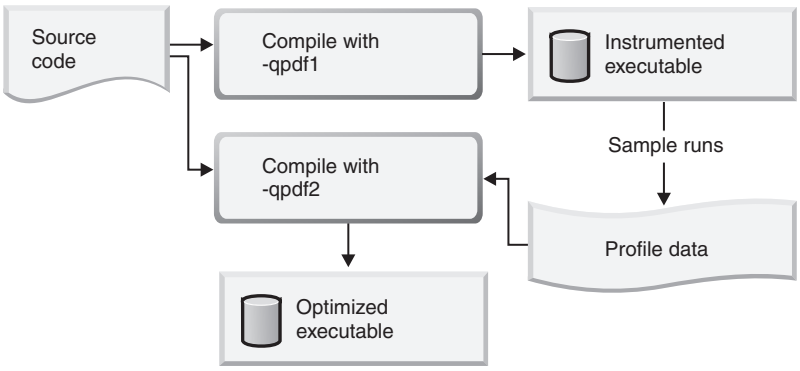
- **-qipa**, **-Q**, and **-qlist** in the *XL C/C++ Compiler Reference*

Using profile-directed feedback

You can use profile-directed feedback (PDF) to tune the performance of your application for a typical usage scenario. The compiler optimizes the application based on an analysis of how often branches are taken and blocks of code are executed. The PDF process is intended to be used after other debugging and tuning is finished, as one of the last steps before putting the application into production. Other optimizations such as **-qipa** and optimization levels **-O4** and **-O5** can also benefit when used in conjunction with PDF.

The following diagram illustrates the PDF process.

Figure 2. Profile-directed feedback



You first compile the program with the **-qpdf1** option (with a minimum optimization level of **-O2**), which generates profile data by using the compiled program in the same ways that users will typically use it. You then compile the program again, with the **-qpdf2** option. This optimizes the program based on the profile data. Alternatively, if you want to save considerable time by avoiding a full recompilation in the **-qpdf2** step, you can simply relink the object files produced by the **-qpdf1** step.

To use PDF, follow these steps:

1. Compile some or all of the source files in a program with the **-qpdf1** option. You need to specify at least the **-O2** optimizing option and you also need to link with at least **-O2** in effect. Note the compiler options that you use to compile the files; you will need to use the same options later.
2. Run the program all the way through using data that is representative of the data that will be used during a normal run of your finished program. The program records profiling information when it finishes. You can run the program multiple times with different data sets, and the profiling information is accumulated to provide a count of how often branches are taken and blocks of code are executed, based on the input data used. When the application exits, by default, it writes profiling information to the PDF file in the current working directory or the directory specified by the PDFDIR environment variable. The default name for the instrumentation file is ._pdf . To override the defaults, use the **-qipa=pdfname** option in the **-qpdf1** step.
3. Recompile your program using the same compiler options as before, but change **-qpdf1** to **-qpdf2**. In this second compilation, the accumulated profiling information is used to fine-tune the optimizations. The resulting program contains no profiling overhead and runs at full speed.

   Note: The options**-L**, **-l**, and some others are linker options, and you can change them at this point.

As an intermediate step, you can use **-qpdf2** to link the object files created by the **-qpdf1** pass without recompiling the source on the **-qpdf2** pass. This can save considerable time and help fine tune large applications for optimization. You can create and test different flavors of PDF optimized binaries by using different options on the **-qpdf2** pass.

Notes:

- You do not need to compile all of the application's code with the **-qpdf1** option to benefit from the PDF process. In a large application, you might want to concentrate on those areas of the code that can benefit most from optimization.

- When compiling your program with **-qpdf1** or **-qpdf2**, by default, the **-qipa** option is also invoked with **level=0**
- To avoid wasting compilation and execution time, make sure that the PDFDIR environment variable is set to an absolute path. Otherwise, you might run the application from the wrong directory, and it will not be able to locate the profile data files. When that happens, the program may not be optimized correctly or may be stopped by a segmentation fault. A segmentation fault might also happen if you change the value of the PDFDIR variable and execute the application before finishing the PDF process.
- You must use the same set of compiler options at all compilation steps for a particular program. Otherwise, PDF cannot optimize your program correctly and may even slow it down. All compiler settings must be the same, including any supplied by configuration files.
- Avoid mixing PDF files created by the current version level of XL C/C++ with PDF files created by other version levels of the compiler.
- If you compile a program with **-qpdf1**, remember that it will generate profiling information when it runs, which involves some performance overhead. This overhead goes away when you recompile with **-qpdf2** or with no PDF at all.

You can take more control of the PDF file generation, as follows:

1. Compile some or all of the source files in the application with **-qpdf1** and a minimum of **-O2**.
2. Run the application using a typical data set or several typical data sets. By default, this produces a PDF file in the current directory. The default name of the PDF file is ._pdf.
3. Change the PDF file location specified by the PDFDIR environment variable or the **-qipa=pdfname** option to produce a PDF file in a different location.
4. Recompile or relink the application with **-qpdf1** and a minimum of **-O2**.
5. Repeat steps 3 and 4 as often as you want.
6. Use the **mergepdf** utility to combine the PDF files into one PDF file. For example, if you produce three PDF files that represent usage patterns that will occur 53%, 32%, and 15% of the time respectively, you can use this command:

   ```
   mergepdf -r 53 path1  -r 32 path2  -r 15 path3
   ```

7. Recompile or relink the application with **-qpdf2** and a minimum of **-O**.

To erase the information in the PDF directory, use the **cleanpdf** utility or the **resetpdf** utility.

Viewing profiling information with showpdf

To collect and view detailed information on function call and block statistics, compile with the **-qshowpdf** option and then use the **showpdf** utility. The following example shows how you can use profile-directed feedback (PDF) with the **showpdf** utility to view the call and block statistics for a "Hello World" application.

The source for the program file `hello.c` is as follows:

```
#include <stdio.h>
void HelloWorld()
{
printf("Hello World");
}
main()
```

```
{
HelloWorld();
return 0;
}
```

1. Compile the source file.

   ```
   xlc -qpdf1 -qshowpdf -O hello.c
   ```

2. Run the resulting executable program **a.out** using a typical data set or several typical data sets.
3. Run the **showpdf** utility to display the call and block counts for the executable file. If you used the **-qipa=pdfname** option during compilation, use the **-f** option to indicate the instrumentation file.

   ```
   showpdf -f instr1
   ```

The results will look similar to this:

```
HelloWorld(4):  1 (hello.c)

Call Counters:
5 | 1  printf(6)

Call coverage = 100% ( 1/1 )

Block Counters:
3-5 | 1
6 |
6 | 1

Block coverage = 100% ( 2/2 )

----------------------------------
main(5):  1 (hello.c)

Call Counters:
10 | 1  HelloWorld(4)

Call coverage = 100% ( 1/1 )

Block Counters:
8-11 | 1
11 |

Block coverage = 100% ( 1/1 )

Total Call coverage = 100% ( 2/2 )
Total Block coverage = 100% ( 3/3 )
```

Related information

- **-qpdf** and **-showpdf** in the *XL C/C++ Compiler Reference*

Object level profile-directed feedback

In addition to optimizing entire executables, profile-directed feedback (PDF) can also be applied to specific objects. This can be an advantage in applications where patches or updates are distributed as object files or libraries rather than as executables. Also, specific areas of functionality in your application can be optimized without you needing to go through the process of relinking the entire application. In large applications, you can save the time and trouble that otherwise would have been spent relinking the application.

The process for using object level PDF is essentially the same as the standard PDF process but with a small change to the **-qpdf2** step. For object level PDF, compile

your application using **-qpdf1**, execute the application with representative data, compile the application again with **-qpdf2** but now also use the **-qnoipa** option so that the linking step is skipped.

The steps below outline this process:

1. Compile your application using **-qpdf1**. For example:
   ```
   xlc -c -O3 -qpdf1 file1.c file2.c file3.c
   ```
   In this example, we are using the option **-O3** to indicate that we want a moderate level of optimization.
2. Link the object files to get an instrumented executable.
   ```
   xlc -O3 -qpdf1 file1.o file2.o file3.o
   ```
   Note: you must use the same optimization options. In this example, the optimization option **-O3**.
3. Run the instrumented executable with sample data that is representative of the data you want to optimize for.
   ```
   a.out < sample_data
   ```
4. Compile the application again using **-qpdf2**. Specify the **-qnoipa** option so that the linking step is skipped and PDF optimization is applied to the object files rather than to the entire executable. **Note**: you must use the same optimization options as in the previous steps. In this example, the optimization option **-O3**.
   ```
   xlc -c -O3 -qpdf2 -qnoipa file1.c file2.c file3.c
   ```
   The resulting output of this step are object files optimized for the sample data processed by the original instrumented executable. In this example, the optimized object files would be file1.o, file2.o, and file3.o. These can be linked using the system loader **ld** or by omitting the **-c** option in the **-qpdf2** step.

Notes:

- If you want to specify a file name for the profile that is created, use the **pdfname** suboption in both the **-qpdf1** and **-qpdf2** steps. For example:
  ```
  xlc -O3 -qpdf1=pdfname=myprofile file1.c file2.c file3.c
  ```
  Without the **pdfname** suboption, by default the file name will be ._pdf; the location of the file will be the current working directory or whatever directory you have set using the PDFDIR environment variable.
- You must use the same optimization options in each compilation and linking step.
- Because **-qnoipa** needs to be specified in the **-qpdf2** step so that linking of your object files is skipped, you will not be able to use interprocedural analysis (IPA) optimizations and object level PDF at the same time.

Other optimization options

The following options are available to control particular aspects of optimization. They are often enabled as a group or given default values when you enable a more general optimization option or level. For more information on these options, see the heading for each option in the *XL C/C++ Compiler Reference XL C/C++ Compiler Reference*.

Table 17. Selected compiler options for optimizing performance

Option	Description
-qignerrno	Allows the compiler to assume that `errno` is not modified by library function calls, so that such calls can be optimized. Also allows optimization of square root operations, by generating inline code rather than calling a library function. (For processors that support `sqrt`.)
-qsmallstack	Instructs the compiler to compact stack storage. Doing so may increase heap usage.
-qinline	Controls inlining by the low-level optimizer.
-qunroll	Independently controls loop unrolling. **-qunroll** is implicitly activated under **-O3**.
-qtbtable	Controls the generation of traceback table information. 64-bit mode only.
C++ **-qnoeh**	Informs the compiler that no C++ exceptions will be thrown and that cleanup code can be omitted. If your program does not throw any C++ exceptions, use this option to compact your program by removing exception-handling code.
-qnounwind	Informs the compiler that the stack will not be unwound while any routine in this compilation is active. This option can improve optimization of non-volatile register saves and restores. In C++, the **-qnounwind** option implies the **-qnoeh** option.
-qnostrict	Allows the compiler to reorder floating-point calculations and potentially excepting instructions. A potentially excepting instruction is one that might raise an interrupt due to erroneous execution (for example, floating-point overflow, a memory access violation). **-qnostrict** is used by default for optimization levels **-O3** and higher.

Chapter 8. Debugging optimized code

Debugging optimized programs presents special usability problems. Optimization can change the sequence of operations, add or remove code, change variable data locations, and perform other transformations that make it difficult to associate the generated code with the original source statements. For example:

Data location issues
With an optimized program, it is not always certain where the most current value for a variable is located. For example, a value in memory may not be current if the most current value is being stored in a register. Most debuggers are incapable of following the removal of stores to a variable, and to the debugger it appears as though that variable is never updated, or possibly even set. This contrasts with no optimization where all values are flushed back to memory and debugging can be more effective and usable.

Instruction scheduling issues
With an optimized program, the compiler may reorder instructions. That is, instructions may not be executed in the order the programmer would expect based on the sequence of lines in their original source code. Also, the sequence of instructions may not be contiguous. As the user steps through their program with a debugger, it may appear as if they are returning to a previously executed line in their code (interleaving of instructions).

Consolidating variable values
Optimizations can result in the removal and consolidation of variables. For example, if a program has two expressions that assign the same value to two different variables, the compiler may substitute a single variable. This can inhibit debug usability because a variable that a programmer is expecting to see is no longer available in the optimized program.

There are a couple of different approaches you can take to improve debug capabilities while also optimizing your program:

Debug non-optimized code first
Debug a non-optimized version of your program first, then recompile it with your desired optimization options. See “Debugging before optimization” on page 52 for some compiler options that are useful in this approach.

Use -qoptdebug
When compiling with **-O3** optimization or higher, use the compiler option **-qoptdebug** to generate a pseudocode file that more accurately maps to how instructions and variable values will operate in an optimized program. With this option, when you load your program into a debugger, you will be debugging the pseudocode for the optimized program. See “Using -qoptdebug to help debug optimized programs” on page 53 for more information.

Understanding different results in optimized programs

Here are some reasons why an optimized program might produce different results from one that has not undergone the optimization process:

- Optimized code can fail if a program contains code that is not valid. The optimization process relies on your application conforming to language standards.
- If a program that works without optimization fails when you optimize, check the cross-reference listing and the execution flow of the program for variables that are used before they are initialized. Compile with the **-qinitauto=hex_value** option to try to produce the incorrect results consistently. For example, using **-qinitauto=FF** gives variables an initial value of ″negative not a number″ (-NAN). Any operations on these variables will also result in NAN values. Other bit patterns (***hex_value***) may yield different results and provide further clues as to what is going on. Programs with uninitialized variables can appear to work properly when compiled without optimization, because of the default assumptions the compiler makes, but can fail when you optimize. Similarly, a program can appear to execute correctly after optimization, but fails at lower optimization levels or when run in a different environment.
- A variation on uninitialized storage. Referring to an automatic-storage variable by its address after the owning function has gone out of scope leads to a reference to a memory location that can be overwritten as other auto variables come into scope as new functions are called.

Use with caution debugging techniques that rely on examining values in storage. The compiler might have deleted or moved a common expression evaluation. It might have assigned some variables to registers, so that they do not appear in storage at all.

Debugging before optimization

First debug your program, then recompile it with your desired optimization options, and test the optimized program before placing the program into production. If the optimized code does not produce the expected results, you can attempt to isolate the specific optimization problems in a debugging session.

The following list presents options that provide specialized information, which can be helpful during the development of optimized code:

-qsmp=noopt If you are debugging SMP code, **-qsmp=noopt** ensures that the compiler performs only the minimum transformations necessary to parallelize your code and preserves maximum debug capability.

-qkeepparm Ensures that procedure parameters are stored on the stack even during optimization. This can negatively impact execution performance. The **-qkeepparm** option then provides access to the values of incoming parameters to tools, such as debuggers, simply by preserving those values on the stack.

-qlist Instructs the compiler to emit an object listing. The object listing includes hex and pseudo-assembly representations of the generated instructions, traceback tables, and text constants.

-qreport Instructs the compiler to produce a report of the loop transformations it performed and how the program was parallelized. For **-qreport** to generate a listing, the options **-qhot** or **-qsmp** should also be specified.

-qinitauto Instructs the compiler to emit code that initializes all automatic variables to a given value.

-qipa=list Instructs the compiler to emit an object listing that provides information for IPA optimization.

You can also use the **snapshot** pragma to ensure to that certain variables are visible to the debugger at points in your application.

Using -qoptdebug to help debug optimized programs

The purpose of the **-qoptdebug** compiler option is to aid the debugging of optimized programs. It does this by creating pseudocode that maps more closely to the instructions and values of an optimized program than the original source code. When a program compiled with this option is loaded into a debugger, you will be debugging the pseudocode rather than your original source. By making optimizations explicit in pseudocode, you can gain a better understanding of how your program is really behaving under optimization. Files containing the pseudocode for your program will be generated with the file suffix `.optdbg`. Only line debugging is supported for this feature.

Compile your program as in the following example:

```
xlc myprogram.c -O3 -qhot -g -qoptdebug
```

In this example, your source file will be compiled to a.out. The pseudocode for the optimized program will be written to a file called myprogram.optdbg which can be referred to while debugging your program.

Notes:

- The **-g** or the **-qlinedebug** option must also be specified in order for the compiled executable to be debuggable. However, if neither of these options are specified, the pseudocode file `<output_file>.optdbg` containing the optimized pseudocode will still be generated.
- The **-qoptdebug** option only has an effect when one or more of the optimization options **-qhot**, **-qsmp**, **-qipa**, or **-qpdf** are specified, or when the optimization levels that imply these options are specified; that is, the optimization levels **-O3**, **-O4**, and **-O5**. The example shows the optimization options **-qhot** and **-O3**.

Debugging the optimized program

See the figures below as an aid to understanding how the compiler may apply optimizations to a simple program and how debugging it would differ from debugging your original source.

Figure 3 on page 54 **Original code**: Represents the original non-optimized code for a simple program. It presents a couple of optimization opportunities to the compiler. For example, the variables `z` and `d` are both assigned by the equivalent expressions `x + y`. Therefore, these two variables can be consolidated in the optimized source. Also, the loop can be unrolled. In the optimized source, you would see iterations of the loop listed explicitly.

Figure 4 on page 54 **dbx debugger listing**: Represents a listing of the optimized source as shown in the dbx debugger. Note the unrolled loop and the consolidation of values assigned by the `x + y` expression.

Figure 5 on page 55 **Stepping through optimized source**: Shows an example of stepping through the optimized source using the dbx debugger. Note, there is no

longer a correspondence between the line numbers for these statements in the optimized source as compared to the line numbers in the original source.

```
#include "stdio.h"

void foo(int x, int y, char* w)
{
 char* s = w+1;
 char* t = w+1;
 int z = x + y;
 int d = x + y;
 int a = printf("TEST\n");

 for (int i = 0; i < 4; i++)
  printf("%d %d %d %s %s\n", a, z, d, s, t);
 }

int main()
{
 char d[] = "DEBUG";
 foo(3, 4, d);
 return 0;
}
```

Figure 3. **Original code**

```
dbx> list
     1    3 |  void foo(long x, long y, char * w)
     2    9 |  {
     3           a = printf("TEST/n");
     4   12 |    @CSE0 = x + y;
     5           printf("%d %d %d %s %s/n",a,@CSE0,@CSE0,((char *)w  + 1),((char *)w  + 1));
     6           printf("%d %d %d %s %s/n",a,@CSE0,@CSE0,((char *)w  + 1),((char *)w  + 1));
     7           printf("%d %d %d %s %s/n",a,@CSE0,@CSE0,((char *)w  + 1),((char *)w  + 1));
     8           printf("%d %d %d %s %s/n",a,@CSE0,@CSE0,((char *)w  + 1),((char *)w  + 1));
     9   13 |    return;
    10         } /* function */
    11   15 |  long main()
    12   17 |  {
    13           d$init$0 = "DEBUG";
    14   18 |    foo(3,4,&d)
    15   19 |    rstr = 0;
    16           return rstr;
    17   20 |  } /* function */
```

Figure 4. **dbx debugger listing**

```
dbx> stop at 3
[1] stop at "myprogram.o.optdbg":3
dbx> run
TEST
[1] stopped in foo(int,int,char*) at line 3 in file "myprogram.o.optdbg" ($t1)
    3       16 |    @CSE0 = x + y;
dbx> step
stopped in foo(int,int,char*) at line 4 in file "myprogram.o.optdbg" ($t1)
    4               printf("%d %d %d %s %s/n",a,@CSE0,@CSE0,((char *)w  + 1),((char *)w  + 1));
dbx> step
3 7 7 EBUG EBUG
stopped in foo(int,int,char*) at line 5 in file "myprogram.o.optdbg" ($t1)
    5               printf("%d %d %d %s %s/n",a,@CSE0,@CSE0,((char *)w  + 1),((char *)w  + 1));
dbx> cont
3 7 7 EBUG EBUG
3 7 7 EBUG EBUG
3 7 7 EBUG EBUG

execution completed
```

Figure 5. **Stepping through optimized source**

Chapter 9. Coding your application to improve performance

Chapter 7, "Optimizing your applications," on page 33 discusses the various compiler options that XL C/C++ provides for optimizing your code with minimal coding effort. If you want to take your application a step further, to complement and take the most advantage of compiler optimizations, the following sections discuss C and C++ programming techniques that can improve performance of your code:

Find faster input/output techniques

There are a number of ways to improve your program's performance of input and output:

- Use binary streams instead of text streams. In binary streams, data is not changed on input or output.
- Use the low-level I/O functions, such as `open` and `close`. These functions are faster and more specific to the application than the stream I/O functions like `fopen` and `fclose`. You must provide your own buffering for the low-level functions.
- If you do your own I/O buffering, make the buffer a multiple of 4K, which is the size of a page.
- When reading input, read in a whole line at once rather than one character at a time.
- If you know you have to process an entire file, determine the size of the data to be read in, allocate a single buffer to read it to, read the whole file into that buffer at once using `read`, and then process the data in the buffer. This reduces disk I/O, provided the file is not so big that excessive swapping will occur. Consider using the `mmap` function to access the file.
- Instead of `scanf` and `fscanf`, use `fgets` to read in a string, and then use one of `atoi`, `atol`, `atof`, or `_atold` to convert it to the appropriate format.
- Use `sprintf` only for complicated formatting. For simpler formatting, such as string concatenation, use a more specific string function.

Reduce function-call overhead

When you write a function or call a library function, consider the following guidelines:

- Call a function directly, rather than using function pointers.
- Pass a value to a function as an argument, rather than letting the function take the value from a global variable.

- Use constant arguments in inlined functions whenever possible. Functions with constant arguments provide more opportunities for optimization.
- Use the **#pragma expected_value** preprocessor directive so that the compiler can optimize for common values used with a function.
- Use the **#pragma isolated_call** preprocessor directive to list functions that have no side effects and do not depend on side effects.
- Use **#pragma disjoint** within functions for pointers or reference parameters that can never point to the same memory.
- Declare a nonmember function as static whenever possible. This can speed up calls to the function.
- C++ Usually, you should not declare all your virtual functions inline. If all virtual functions in a class are inline, the virtual function table and all the virtual function bodies will be replicated in each compilation unit that uses the class.
- C++ When declaring functions, use the `const` specifier whenever possible.
- C Fully prototype all functions. A full prototype gives the compiler and optimizer complete information about the types of the parameters. As a result, promotions from unwidened types to widened types are not required, and parameters can be passed in appropriate registers.
- C Avoid using unprototyped variable argument functions.
- Design functions so that the most frequently used parameters are in the leftmost positions in the function prototype.
- Avoid passing by value structures or unions as function parameters or returning a structure or a union. Passing such aggregates requires the compiler to copy and store many values. This is worse in C++ programs in which class objects are passed by value because a constructor and destructor are called when the function is called. Instead, pass or return a pointer to the structure or union, or pass it by reference.
- Pass non-aggregate types such as `int` and `short` by value rather than passing by reference, whenever possible.
- If your function exits by returning the value of another function with the same parameters that were passed to your function, put the parameters in the same order in the function prototypes. The compiler can then branch directly to the other function.
- Use the built-in functions, which include string manipulation, floating-point, and trigonometric functions, instead of coding your own. Intrinsic functions require less overhead and are faster than a function call, and often allow the compiler to perform better optimization.

  C++ Your functions are automatically mapped to built-in functions if you include the XL C/C++ header files.

  C Your functions are mapped to built-in functions if you include `math.h` and `string.h`.
- Selectively mark your functions for inlining, using the `inline` keyword. An inlined function requires less overhead and is generally faster than a function call. The best candidates for inlining are small functions that are called frequently from a few places, or functions called with one or more compile-time constant parameters, especially those that affect `if`, `switch` or `for` statements. You might also want to put these functions into header files, which allows automatic inlining across file boundaries even at low optimization levels. Be sure to inline all functions that only load or store a value, or use simple operators

such as comparison or arithmetic operators. Large functions and functions that are called rarely might not be good candidates for inlining.

- Avoid breaking your program into too many small functions. If you must use small functions, seriously consider using the **-qipa** compiler option, which can automatically inline such functions, and uses other techniques for optimizing calls between functions.
- C++ Avoid virtual functions and virtual inheritance unless required for class extensibility. These language features are costly in object space and function invocation performance.

Related information

- **#pragma isolated_call**, **#pragma disjoint**, and **-qipa** in the *XL C/C++ Compiler Reference*

Manage memory efficiently

Because C++ objects are often allocated from the heap and have limited scope, memory use affects performance more in C++ programs than it does in C programs. For that reason, consider the following guidelines when you develop C++ applications:

- In a structure, declare the largest members first.
- In a structure, place variables near each other if they are frequently used together.
- C++ Ensure that objects that are no longer needed are freed or otherwise made available for reuse. One way to do this is to use an *object manager*. Each time you create an instance of an object, pass the pointer to that object to the object manager. The object manager maintains a list of these pointers. To access an object, you can call an object manager member function to return the information to you. The object manager can then manage memory usage and object reuse.
- Storage pools are a good way of keeping track of used memory (and reclaiming it) without having to resort to an object manager or reference counting.
- C++ Avoid copying large, complicated objects.
- C++ Avoid performing a *deep copy* if a *shallow copy* is all you require. For an object that contains pointers to other objects, a shallow copy copies only the pointers and not the objects to which they point. The result is two objects that point to the same contained object. A deep copy, however, copies the pointers and the objects they point to, as well as any pointers or objects contained within that object, and so on.
- C++ Use virtual methods only when absolutely necessary.

Optimize variables

Consider the following guidelines:

- Use local variables, preferably automatic variables, as much as possible.

  The compiler must make several worst-case assumptions about a global variable. For example, if a function uses external variables and also calls external functions, the compiler assumes that every call to an external function could change the value of every external variable. If you know that a global variable is not affected by any function call, and this variable is read several times with function calls interspersed, copy the global variable to a local variable and then use this local variable.

- If you must use global variables, use static variables with file scope rather than external variables whenever possible. In a file with several related functions and static variables, the optimizer can gather and use more information about how the variables are affected.
- If you must use external variables, group external data into structures or arrays whenever it makes sense to do so. All elements of an external structure use the same base address.
- The **#pragma isolated_call** preprocessor directive can improve the runtime performance of optimized code by allowing the compiler to make less pessimistic assumptions about the storage of external and static variables. Isolated call functions with constant or loop-invariant parameters can be moved out of loops, and multiple calls with the same parameters can be replaced with a single call.
- Avoid taking the address of a variable. If you use a local variable as a temporary variable and must take its address, avoid reusing the temporary variable. Taking the address of a local variable inhibits optimizations that would otherwise be done on calculations involving that variable.
- Use constants instead of variables where possible. The optimizer will be able to do a better job reducing runtime calculations by doing them at compile-time instead. For instance, if a loop body has a constant number of iterations, use constants in the loop condition to improve optimization (`for (i=0; i<4; i++)` can be better optimized than `for (i=0; i<x; i++)`).
- Use register-sized integers (`long` data type) for scalars. For large arrays of integers, consider using one- or two-byte integers or bit fields.
- Use the smallest floating-point precision appropriate to your computation.

Related information

- **#pragma isolated_call** in *XL C/C++ Compiler Reference*

Manipulate strings efficiently

The handling of string operations can affect the performance of your program.

- When you store strings into allocated storage, align the start of the string on an 8-byte boundary.
- Keep track of the length of your strings. If you know the length of a string, you can use `mem` functions instead of `str` functions. For example, `memcpy` is faster than `strcpy` because it does not have to search for the end of the string.
- If you are certain that the source and target do not overlap, use `memcpy` instead of `memmove`. This is because `memcpy` copies directly from the source to the destination, while `memmove` might copy the source to a temporary location in memory before copying to the destination (depending on the length of the string).
- When manipulating strings using `mem` functions, faster code will be generated if the *count* parameter is a constant rather than a variable. This is especially true for small count values.
- Make string literals read-only, whenever possible. This improves certain optimization techniques and reduces memory usage if there are multiple uses of the same string. You can explicitly set strings to read-only by using **#pragma strings (readonly)** in your source files or **-qro** (this is enabled by default) to avoid changing your source files.

Related information

- **#pragma strings (readonly)** and **-qro** in the *XL C/C++ Compiler Reference*

Optimize expressions and program logic

Consider the following guidelines:

- If components of an expression are used in other expressions, assign the duplicated values to a local variable.
- Avoid forcing the compiler to convert numbers between integer and floating-point internal representations. For example:

```
float array[10];
float x = 1.0;
int i;
for (i = 0; i< 9; i++)  {     /* No conversions needed */
     array[i] = array[i]*x;
     x = x + 1.0;
    }
for (i = 0; i< 9; i++)  {       /* Multiple conversions needed */
     array[i] = array[i]*i;
    }
```

  When you must use mixed-mode arithmetic, code the integer and floating-point arithmetic in separate computations whenever possible.
- Avoid `goto` statements that jump into the middle of loops. Such statements inhibit certain optimizations.
- Improve the predictability of your code by making the fall-through path more probable. Code such as:

```
if (error) {handle error} else {real code}
```

  should be written as:

```
if (!error) {real code} else {error}
```

- If one or two cases of a `switch` statement are typically executed much more frequently than other cases, break out those cases by handling them separately before the `switch` statement.
- C++ Use `try` blocks for exception handling only when necessary because they can inhibit optimization.
- Keep array index expressions as simple as possible.

Optimize operations in 64-bit mode

The ability to handle larger amounts of data directly in physical memory rather than relying on disk I/O is perhaps the most significant performance benefit of 64-bit machines. However, some applications compiled in 32-bit mode perform better than when they are recompiled in 64-bit mode. Some reasons for this include:

- 64-bit programs are larger. The increase in program size places greater demands on physical memory.
- 64-bit long division is more time-consuming than 32-bit integer division.
- 64-bit programs that use 32-bit signed integers as array indexes might require additional instructions to perform sign extension each time the array is referenced.

Some ways to compensate for the performance liabilities of 64-bit programs include:

- Avoid performing mixed 32- and 64-bit operations. For example, adding a 32-bit data type to a 64-bit data type requires that the 32-bit type be sign-extended to clear the upper 32 bits of the register. This slows the computation.

- Use `long` types instead of `signed`, `unsigned`, and plain `int` types for variables which will be frequently accessed, such as loop counters and array indexes. Doing so frees the compiler from having to truncate or sign-extend array references, parameters during function calls, and function results during returns.

Chapter 10. Using the high performance libraries

IBM XL C/C++ Advanced Edition for Linux, V9.0 is shipped with a set of libraries for high-performance mathematical computing:

- The Mathematical Acceleration Subsystem (MASS) is a set of libraries of tuned mathematical intrinsic functions that provide improved performance over the corresponding standard system math library functions. MASS is described in "Using the Mathematical Acceleration Subsystem libraries (MASS) ."
- The Basic Linear Algebra Subprograms (BLAS) are a set of routines which provide matrix/vector multiplication functions tuned for PowerPC architectures. The BLAS functions are described in "Using the Basic Linear Algebra Subprograms (BLAS)" on page 71.

Using the Mathematical Acceleration Subsystem libraries (MASS)

The MASS libraries consist of a library of scalar XL C/C++ functions described in "Using the scalar library"; and a set of vector libraries tuned for specific architectures, described in "Using the vector libraries" on page 66. The functions contained in both scalar and vector libraries are automatically called at certain levels of optimization, but you can also call them explicitly in your programs. Note that the accuracy and exception handling might not be identical in MASS functions and system library functions.

"Compiling and linking a program with MASS" on page 71 describes how to compile and link a program that uses the MASS libraries, and how to selectively use the MASS scalar library functions in concert with the regular system library scalar functions.

Note: On Linux, 32-bit and 64-bit objects cannot be combined in the same library, so two versions of the scalar and vector libraries are shipped with the compiler: libmass.a and libmassv.a for 32-bit applications and libmass_64.a and libmassv_64.a for 64-bit applications.

Using the scalar library

The MASS scalar libraries libmass.a (32-bit) and libmass_64.a (64-bit) contain an accelerated set of frequently used math intrinsic functions that provide improved performance over the corresponding standard system library functions. These functions are available when you compile programs with any of the following options:

- **-qhot -qignerrno -qnostrict**
- **-qhot -O3**
- **-O4**
- **-O5**

the compiler automatically uses the faster MASS functions for most math library functions. In fact, the compiler first tries to ″vectorize″ calls to math library functions by replacing them with the equivalent MASS vector functions; if it cannot do so, it uses the MASS scalar functions. When the compiler performs this automatic replacement of math library functions, it uses versions of the MASS functions contained in the system library `libxlopt.a`. You do not need to add any special calls to the MASS functions in your code, or to link to the `libxlopt` library.

If you are not using any of the optimization options listed above, and want to explicitly call the MASS scalar functions, you can do so as follows:

1. Provide the prototypes for the functions (except `anint`, `cosisin`, `dnint`, `sincos`, and `rsqrt`), by including `math.h` in your source files.
2. Provide the prototypes for `anint`, `cosisin`, `dnint`, `sincos`, and `rsqrt`, by including `mass.h` in your source files.
3. Link the MASS scalar library libmass.a (or the 64-bit version, libmass_64.a) with your application. For instructions, see "Compiling and linking a program with MASS" on page 71.

The MASS scalar functions accept double-precision parameters and return a double-precision result, or accept single-precision parameters and return a single-precision result, except `sincos` which gives 2 double-precision results. They are summarized in Table 18.

Table 18. MASS scalar functions

Double-precision function	**Single-precision function**	**Description**	**Double-precision function prototype**	**Single-precision function prototype**
acos	acosf	Returns the arccosine of `x`	double acos (double x);	float acosf (float x);
acosh	acoshf	Returns the hyperbolic arccosine of `x`	double acosh (double x);	float acoshf (float x);
	anint	Returns the rounded integer value of `x`		float anint (float x);
asin	asinf	Returns the arcsine of `x`	double asin (double x);	float asinf (float x);
asinh	asinhf	Returns the hyperbolic arcsine of `x`	double asinh (double x);	float asinhf (float x);
atan2	atan2f	Returns the arctangent of `x/y`	double atan2 (double x, double y);	float atan2f (float x, float y);
atan	atanf	Returns the arctangent of `x`	double atan (double x);	float atanf (float x);
atanh	atanhf	Returns the hyperbolic arctangent of `x`	double atanh (double x);	float atanhf (float x);
cbrt	cbrtf	Returns the cube root of `x`	double cbrt (double x);	float cbrtf (float x);
copysign	copysignf	Returns `x` with the sign of `y`	double copysign (double x,double y);	float copysignf (float x);
cos	cosf	Returns the cosine of `x`	double cos (double x);	float cosf (float x);
cosh	coshf	Returns the hyperbolic cosine of `x`	double cosh (double x);	float coshf (float x);

Table 18. MASS scalar functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
cosisin		Returns a complex number with the real part the cosine of x and the imaginary part the sine of x.	double_Complex cosisin (double);	
dnint		Returns the nearest integer to x (as a double)	double dnint (double x);	
erf	erff	Returns the error function of x	double erf (double x);	float erff (float x);
erfc	erfcf	Returns the complementary error function of x	double erfc (double x);	float erfcf (float x);
exp	expf	Returns the exponential function of x	double exp (double x);	float expf (float x);
expm1	expm1f	Returns (the exponential function of x) – 1	double expm1 (double x);	float expm1f (float x);
hypot	hypotf	Returns the square root of $x^2 + y^2$	double hypot (double x, double y);	float hypotf (float x, float y);
lgamma	lgammaf	Returns the natural logarithm of the absolute value of the Gamma function of x	double lgamma (double x);	float lgammaf (float x);
log	logf	Returns the natural logarithm of x	double log (double x);	float logf (float x);
log10	log10f	Returns the base 10 logarithm of x	double log10 (double x);	float log10f (float x);
log1p	log1pf	Returns the natural logarithm of (x + 1)	double log1p (double x);	float log1pf (float x);
pow	powf	Returns x raised to the power y	double pow (double x, double y);	float powf (float x);
rsqrt		Returns the reciprocal of the square root of x	double rsqrt (double x);	
sin	sinf	Returns the sine of x	double sin (double x);	float sinf (float x);

Table 18. MASS scalar functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
sincos		Sets *s to the sine of x and *c to the cosine of x	void sincos (double x, double* s, double* c);	
sinh	sinhf	Returns the hyperbolic sine of x	double sinh (double x);	float sinhf (float x);
sqrt		Returns the square root of x	double sqrt (double x);	
tan	tanf	Returns the tangent of x	double tan (double x);	float tanf (float x);
tanh	tanhf	Returns the hyperbolic tangent of x	double tanh (double x);	float tanhf (float x);

Notes:

- The trigonometric functions (`sin`, `cos`, `tan`) return NaN (Not-a-Number) for large arguments (where the absolute value is greater than 2^{50}pi).
- In some cases, the MASS functions are not as accurate as the `libm.a` library, and they might handle edge cases differently (`sqrt(Inf)`, for example).

Using the vector libraries

When you compile programs with any of the following options:

- **-qhot -qignerrno -qnostrict**
- **-qhot -O3**
- **-O4**
- **-O5**

the compiler automatically attempts to vectorize calls to system math functions by calling the equivalent MASS vector functions (with the exceptions of functions `vdnint`, `vdint`, `vsincos`, `vssincos`, `vcosisin`, `vscosisin`, `vqdrt`, `vsqdrt`, `vrqdrt`, `vsrqdrt`, `vpopcnt4`, and `vpopcnt8`). For automatic vectorization, the compiler uses versions of the MASS functions contained in the system library `libxlopt.a`. You do not need to add any special calls to the MASS functions in your code, or to link to the libxlopt library.

If you are not using any of the optimization options listed above, and want to explicitly call any of the MASS vector functions, you can do so by including the XL C/C++ header `massv.h` file in your source files and linking your application with the appropriate vector library. (Information on linking is provided in "Compiling and linking a program with MASS" on page 71.)

libmassvp4.a
Contains functions that have been tuned for the POWER4™ architecture. If you are using a PPC970 machine, this library is the recommended choice.

libmassvp5.a
Contains functions that have been tuned for the POWER5™ architecture.

libmassvp6.a
Contains functions that have been tuned for the POWER6™ architecture.

On Linux, 32-bit and 64-bit objects must not be mixed in a single library, so a separate 64-bit version of each vector library is provided: libmassvp4_64.a, libmassvp5_64.a, and libmassvp6_64.a

The single-precision and double-precision floating-point functions contained in the vector libraries are summarized in Table 19. The integer functions contained in the vector libraries are summarized in Table 20 on page 70. Note that in C and C++ applications, only call by reference is supported, even for scalar arguments.

With the exception of a few functions (described below), all of the floating-point functions in the vector libraries accept three parameters:

- a double-precision (for double-precision functions) or single-precision (for single-precision functions) vector output parameter
- a double-precision (for double-precision functions) or single-precision (for single-precision functions) vector input parameter
- an integer vector-length parameter

The functions are of the form

function_name (*y*,*x*,*n*)

where *y* is the target vector, *x* is the source vector, and *n* is the vector length. The parameters *y* and *x* are assumed to be double-precision for functions with the prefix `v`, and single-precision for functions with the prefix `vs`. As examples, the following code:

```
#include <massv.h>

double x[500], y[500];
int n;
n = 500;
...
vexp (y, x, &n);
```

outputs a vector *y* of length 500 whose elements are exp(x[i]), where i=0,...,499.

The functions `vdiv`, `vsincos`, `vpow`, and `vatan2` (and their single-precision versions, `vsdiv`, `vssincos`, `vspow`, and `vsatan2`) take four parameters. The functions `vdiv`, `vpow`, and `vatan2` take the parameters (*z*,*x*,*y*,*n*). The function `vdiv` outputs a vector *z* whose elements are x[i]/y[i], where i=0,..,*n–1. The function `vpow` outputs a vector *z* whose elements are $x[i]^{y[i]}$, where i=0,..,*n–1. The function `vatan2` outputs a vector *z* whose elements are atan(x[i]/y[i]), where i=0,..,*n–1. The function `vsincos` takes the parameters (*y*,*z*,*x*,*n*), and outputs two vectors, *y* and *z*, whose elements are sin(x[i]) and cos(x[i]), respectively.

Table 19. MASS floating-point vector functions

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
vacos	vsacos	Sets `y[i]` to the arc cosine of `x[i]`, for i=0,..,*n–1	void vacos (double y[], double x[], int *n);	void vsacos (float y[], float x[], int *n);

Table 19. MASS floating-point vector functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
vacosh	vsacosh	Sets y[i] to the hyperbolic arc cosine of x[i], for i=0,..,*n–1	void vacosh (double y[], double x[], int *n);	void vsacosh (float y[], float x[], int *n);
vasin	vsasin	Sets y[i] to the arc sine of x[i], for i=0,..,*n–1	void vasin (double y[], double x[], int *n);	void vsasin (float y[], float x[], int *n);
vasinh	vsasinh	Sets y[i] to the hyperbolic arc sine of x[i], for i=0,..,*n–1	void vasinh (double y[], double x[], int *n);	void vsasinh (float y[], float x[], int *n);
vatan2	vsatan2	Sets z[i] to the arc tangent of x[i]/y[i], for i=0,..,*n–1	void vatan2 (double z[], double x[], double y[], int *n);	void vsatan2 (float z[], float x[], float y[], int *n);
vatanh	vsatanh	Sets y[i] to the hyperbolic arc tangent of x[i], for i=0,..,*n–1	void vatanh (double y[], double x[], int *n);	void vsatanh (float y[], float x[], int *n);
vcbrt	vscbrt	Sets y[i] to the cube root of x[i], for i=0,..,*n-1	void vcbrt (double y[], double x[], int *n);	void vscbrt (float y[], float x[], int *n);
vcos	vscos	Sets y[i] to the cosine of x[i], for i=0,..,*n–1	void vcos (double y[], double x[], int *n);	void vscos (float y[], float x[], int *n);
vcosh	vscosh	Sets y[i] to the hyperbolic cosine of x[i], for i=0,..,*n–1	void vcosh (double y[], double x[], int *n);	void vscosh (float y[], float x[], int *n);
vcosisin	vscosisin	Sets the real part of y[i] to the cosine of x[i] and the imaginary part of y[i] to the sine of x[i], for i=0,..,*n–1	void vcosisin (double _Complex y[], double x[], int *n);	void vscosisin (float _Complex y[], float x[], int *n);
vdint		Sets y[i] to the integer truncation of x[i], for i=0,..,*n–1	void vdint (double y[], double x[], int *n);	
vdiv	vsdiv	Sets z[i] to x[i]/y[i], for i=0,..,*n–1	void vdiv (double z[], double x[], double y[], int *n);	void vsdiv (float z[], float x[], float y[], int *n);
vdnint		Sets y[i] to the nearest integer to x[i], for i=0,..,n–1	void vdnint (double y[], double x[], int *n);	
vexp	vsexp	Sets y[i] to the exponential function of x[i], for i=0,..,*n–1	void vexp (double y[], double x[], int *n);	void vsexp (float y[], float x[], int *n);

Table 19. MASS floating-point vector functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
vexpm1	vsexpm1	Sets y[i] to (the exponential function of x[i])-1, for i=0,..,*n–1	void vexpm1 (double y[], double x[], int *n);	void vsexpm1 (float y[], float x[], int *n);
vlog	vslog	Sets y[i] to the natural logarithm of x[i], for i=0,..,*n–1	void vlog (double y[], double x[], int *n);	void vslog (float y[], float x[], int *n);
vlog10	vslog10	Sets y[i] to the base-10 logarithm of x[i], for i=0,..,*n–1	void vlog10 (double y[], double x[], int *n);	void vslog10 (float y[], float x[], int *n);
vlog1p	vslog1p	Sets y[i] to the natural logarithm of (x[i]+1), for i=0,..,*n–1	void vlog1p (double y[], double x[], int *n);	void vslog1p (float y[], float x[], int *n);
vpow	vspow	Sets z[i] to x[i] raised to the power y[i], for i=0,..,*n-1	void vpow (double z[], double x[], double y[], int *n);	void vspow (float z[], float x[], float y[], int *n);
vqdrt	vsqdrt	Sets y[i] to the fourth root of x[i], for i=0,..,*n-1	void vqdrt (double y[], double x[], int *n);	void vsqdrt (float y[], float x[], int *n);
vrcbrt	vsrcbrt	Sets y[i] to the reciprocal of the cube root of x[i], for i=0,..,*n-1	void vrcbrt (double y[], double x[], int *n);	void vsrcbrt (float y[], float x[], int *n);
vrec	vsrec	Sets y[i] to the reciprocal of x[i], for i=0,..,*n–1	void vrec (double y[], double x[], int *n);	void vsrec (float y[], float x[], int *n);
vrqdrt	vsrqdrt	Sets y[i] to the reciprocal of the fourth root of x[i], for i=0,..,*n-1	void vrqdrt (double y[], double x[], int *n);	void vsrqdrt (float y[], float x[], int *n);
vrsqrt	vsrsqrt	Sets y[i] to the reciprocal of the square root of x[i], for i=0,..,*n–1	void vrsqrt (double y[], double x[], int *n);	void vsrsqrt (float y[], float x[], int *n);
vsin	vssin	Sets y[i] to the sine of x[i], for i=0,..,*n–1	void vsin (double y[], double x[], int *n);	void vssin (float y[], float x[], int *n);
vsincos	vssincos	Sets y[i] to the sine of x[i] and z[i] to the cosine of x[i], for i=0,..,*n–1	void vsincos (double y[], double z[], double x[], int *n);	void vssincos (float y[], float z[], float x[], int *n);
vsinh	vssinh	Sets y[i] to the hyperbolic sine of x[i], for i=0,..,*n–1	void vsinh (double y[], double x[], int *n);	void vssinh (float y[], float x[], int *n);

Table 19. MASS floating-point vector functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
vsqrt	vssqrt	Sets y[i] to the square root of x[i], for i=0,..,*n–1	void vsqrt (double y[], double x[], int *n);	void vssqrt (float y[], float x[], int *n);
vtan	vstan	Sets y[i] to the tangent of x[i], for i=0,..,*n-1	void vtan (double y[], double x[], int *n);	void vstan (float y[], float x[], int *n);
vtanh	vstanh	Sets y[i] to the hyperbolic tangent of x[i], for i=0,..,*n–1	void vtanh (double y[], double x[], int *n);	void vstanh (float y[], float x[], int *n);

Integer functions are of the form *function_name* (*x*[], **n*), where x[] is a vector of 4-byte (for vpopcnt4) or 8-byte (for vpopcnt8) numeric objects (integral or floating-point), and *n is the vector length.

Table 20. MASS integer vector library functions

Function	Description	Prototype
vpopcnt4	Returns the total number of 1 bits in the concatenation of the binary representation of x[i], for i=0,..,*n–1 , where x is a vector of 32-bit objects.	unsigned int vpopcnt4 (void *x, int *n)
vpopcnt8	Returns the total number of 1 bits in the concatenation of the binary representation of x[i], for i=0,..,*n–1 , where x is a vector of 64-bit objects.	unsigned int vpopcnt8 (void *x, int *n)

Overlap of input and output vectors

In most applications, the MASS vector functions are called with disjoint input and output vectors; that is, the two vectors do not overlap in memory. Another common usage scenario is to call them with the same vector for both input and output parameters (for example, vsin (y, y, &n)). Other kinds of overlap (where input and output vectors are neither disjoint nor identical) should be avoided, since they may produce unexpected results:

- For calls to vector functions that take one input and one output vector (for example, vsin (y, x, &n)):

  The vectors x[0:n-1] and y[0:n-1] must be either disjoint or identical, or unexpected results may be obtained.
- For calls to vector functions that take two input vectors (for example, vatan2 (y, x1, x2, &n)):

  The previous restriction applies to both pairs of vectors y,x1 and y,x2. That is, y[0:n-1] and x1[0:n-1] must be either disjoint or identical; and y[0:n-1] and x2[0:n-1] must be either disjoint or identical.
- For calls to vector functions that take two output vectors (for example, vsincos (y1, y2, x, &n)):

  The above restriction applies to both pairs of vectors y1,x and y2,x. That is, y1[0:n-1] and x[0:n-1] must be either disjoint or identical; and y2[0:n-1] and x[0:n-1] must be either disjoint or identical. Also, the vectors y1[0:n-1] and y2[0:n-1] must be disjoint.

Consistency of MASS vector functions

All the functions in the MASS vector libraries are consistent, in the sense that a given input value will always produce the same result, regardless of its position in the vector, and regardless of the vector length.

Compiling and linking a program with MASS

To compile an application that calls the functions in the MASS libraries, specify **mass** and **massvp4**, **massvp5**, or **massvp6**) (32-bit), or **mass_64** and **massvp4_64**, **massvp5_64**, or **massvp6_64**) (64-bit) on the **-l** linker option.

For example, if the MASS libraries are installed in the default directory, you could specify one of the following:

```
xlc progc.c -o progc -lmass -lmassvp4
xlc progc.c -o progc -lmass_64 -lmassvp4_64 -q64
```

The MASS functions must run in the default rounding mode and floating-point exception trapping settings.

Using libmass.a with the math system library

If you wish to use the libmass.a (or libmass_64.a) scalar library for some functions and the normal math library libm.a for other functions, follow this procedure to compile and link your program:

1. Use the **ar** command to extract the object files of the desired functions from libmass.a or libmass_64.a. For most functions, the object file name is the function name followed by `.s32.o` (for 32-bit mode) or `.s64.o` (for 64-bit mode).[1] For example, to extract the object file for the `tan` function in 32-bit mode, the command would be:

   ```
   ar -x tan.s32.o libmass.a
   ```

2. Archive the extracted object files into another library:

   ```
   ar -qv libfasttan.a tan.s32.o
   ranlib libfasttan.a
   ```

3. Create the final executable using **xlc**, specifying **-lfasttan** instead of **-lmass**:

   ```
   xlc sample.c -o sample dir_containing_libfasttan -lfasttan
   ```

   This links only the `tan` function from MASS (now in `libfasttan.a`) and the remainder of the math functions from the standard system library.

Exceptions:

1. The `sin` and `cos` functions are both contained in the object files sincos.s32.o and sincos.s64.o. The `cosisin` and `sincos` functions are both contained in the object file cosisin.s32.o.
2. The XL C/C++ `pow` function or XL Fortran ** (exponentiation) operator is contained in the object files dxy.s32.o and dxy.s64.o.

Note: The `cos` and `sin` functions will both be exported if either one is exported. `cosisin` and `sincos` will both be exported if either one is exported.

Using the Basic Linear Algebra Subprograms (BLAS)

Four Basic Linear Algebra Subprograms (BLAS) functions are shipped with XL C/C++ in the `libxlopt` library. The functions consist of the following:

- `sgemv` (single-precision) and `dgemv` (double-precision), which compute the matrix-vector product for a general matrix or its transpose

- `sgemm` (single-precision) and `dgemm` (double-precision), which perform combined matrix multiplication and addition for general matrices or their transposes

Because the BLAS routines are written in Fortran, all parameters are passed to them by reference, and all arrays are stored in column-major order.

Note: Some error-handling code has been removed from the BLAS functions in `libxlopt`, and no error messages are emitted for calls to the these functions.

"BLAS function syntax" describes the prototypes and parameters for the XL C/C++ BLAS functions. The interfaces for these functions are similar to those of the equivalent BLAS functions shipped in IBM's Engineering and Scientific Subroutine Library (ESSL); for more detailed information and examples of usage of these functions, you may wish to consult the *Engineering and Scientific Subroutine Library Guide and Reference*, available at http://publib.boulder.ibm.com/clresctr/windows/public/esslbooks.html.

"Linking the libxlopt library" on page 74 describes how to link to the XL C/C++ `libxlopt` library if you are also using a third-party BLAS library.

BLAS function syntax

The prototypes for the `sgemv` and `dgemv` functions are as follows:

```
void sgemv(const char *trans, int *m, int *n, float *alpha,
     void *a, int *lda, void *x, int *incx,
     float *beta, void *y, int *incy);

void dgemv(const char *trans, int *m, int *n, double *alpha,
     void *a, int *lda, void *x, int *incx,
      double *beta, void *y, int *incy);
```

The parameters are as follows:

trans
is a single character indicating the form of the input matrix *a*, where:
- `'N'` or `'n'` indicates that *a* is to be used in the computation
- `'T'` or `'t'` indicates that the transpose of *a* is to be used in the computation

m represents:
- the number of rows in input matrix *a*
- the length of vector *y*, if `'N'` or `'n'` is used for the *trans* parameter
- the length of vector *x*, if `'T'` or `'t'` is used for the *trans* parameter

The number of rows must be greater than or equal to zero, and less than the leading dimension of the matrix *a* (specified in *lda*)

n represents:
- the number of columns in input matrix *a*
- the length of vector *x*, if `'N'` or `'n'` is used for the *trans* parameter
- the length of vector *y*, if `'T'` or `'t'` is used for the *trans* parameter

The number of columns must be greater than or equal to zero.

alpha
is the scaling constant for matrix *a*

a is the input matrix of `float` (for `sgemv`) or `double` (for `dgemv`) values

lda is the leading dimension of the array specified by *a*. The leading dimension

must be greater than zero. The leading dimension must be greater than or equal to 1 and greater than or equal to the value specified in *m*.

x is the input vector of `float` (for `sgemv`) or `double` (for `dgemv`) values.

incx
is the stride for vector *x*. It can have any value.

beta
is the scaling constant for vector *y*

y is the output vector of `float` (for `sgemv`) or `double` (for `dgemv`) values.

incy
is the stride for vector *y*. It must not be zero.

Note: Vector *y* must have no common elements with matrix *a* or vector *x*; otherwise, the results are unpredictable.

The prototypes for the `sgemm` and `dgemm` functions are as follows:

```
void sgemm(const char *transa, const char *transb,
   int *l, int *n, int *m, float *alpha,
   const void *a, int *lda, void *b, int *ldb,
   float *beta, void *c, int *ldc);
void dgemm(const char *transa, const char *transb,
   int *l, int *n, int *m, double *alpha,
   const void *a, int *lda, void *b, int *ldb,
   double *beta, void *c, int *ldc);
```

The parameters are as follows:

transa
is a single character indicating the form of the input matrix *a*, where:

- `'N'` or `'n'` indicates that *a* is to be used in the computation
- `'T'` or `'t'` indicates that the transpose of *a* is to be used in the computation

transb
is a single character indicating the form of the input matrix *b*, where:

- `'N'` or `'n'` indicates that *b* is to be used in the computation
- `'T'` or `'t'` indicates that the transpose of *b* is to be used in the computation

l represents the number of rows in output matrix *c*. The number of rows must be greater than or equal to zero, and less than the leading dimension of *c*.

n represents the number of columns in output matrix *c*. The number of columns must be greater than or equal to zero.

m represents:

- the number of columns in matrix *a*, if `'N'` or `'n'` is used for the *transa* parameter
- the number of rows in matrix *a*, if `'T'` or `'t'` is used for the *transa* parameter

and:

- the number of rows in matrix *b*, if `'N'` or `'n'` is used for the *transb* parameter
- the number of columns in matrix *b*, if `'T'` or `'t'` is used for the *transb* parameter

m must be greater than or equal to zero.

alpha
is the scaling constant for matrix *a*

a is the input matrix *a* of `float` (for `sgemm`) or `double` (for `dgemm`) values

lda is the leading dimension of the array specified by *a*. The leading dimension must be greater than zero. If *transa* is specified as `'N'` or `'n'`, the leading dimension must be greater than or equal to 1. If *transa* is specified as `'T'` or `'t'`, the leading dimension must be greater than or equal to the value specified in *m*.

b is the input matrix *b* of `float` (for `sgemm`) or `double` (for `dgemm`) values.

ldb is the leading dimension of the array specified by *b*. The leading dimension must be greater than zero. If *transb* is specified as `'N'` or `'n'`, the leading dimension must be greater than or equal to the value specified in *m*. If *transa* is specified as `'T'` or `'t'`, the leading dimension must be greater than or equal to the value specified in *n*.

beta
is the scaling constant for matrix *c*

c is the output matrix *c* of `float` (for `sgemm`) or `double` (for `dgemm`) values.

ldc is the leading dimension of the array specified by *c*. The leading dimension must be greater than zero. If *transb* is specified as `'N'` or `'n'`, the leading dimension must be greater than or equal to 0 and greater than or equal to the value specified in *l*.

Note: Matrix *c* must have no common elements with matrices *a* or *b*; otherwise, the results are unpredictable.

Linking the libxlopt library

By default, the `libxlopt` library is linked with any application you compile with XL C/C++. However, if you are using a third-party BLAS library, but want to use the BLAS routines shipped with `libxlopt`, you must specify the `libxlopt` library before any other BLAS library on the command line at link time. For example, if your other BLAS library is called `libblas.a`, you would compile your code with the following command:

```
xlc app.c -lxlopt -lblas
```

The compiler will call the `sgemv`, `dgemv`, `sgemm`, and `dgemm` functions from the `libxlopt` library, and all other BLAS functions in the `libblas.a` library.

Chapter 11. Parallelizing your programs

The compiler offers you three methods of implementing shared memory program parallelization. These are:

- Automatic parallelization of countable program loops, which are defined in "Countable loops." An overview of the compiler's automatic parallelization capabilities is provided in "Enabling automatic parallelization" on page 77.
- Explicit parallelization of C and C++ program code using pragma directives compliant to the OpenMP Application Program Interface specification. An overview of the OpenMP directives is provided in "Using OpenMP directives" on page 77.

All methods of program parallelization are enabled when the **-qsmp** compiler option is in effect without the **omp** suboption. You can enable strict OpenMP compliance with the **-qsmp=omp** compiler option, but doing so will disable automatic parallelization.

Note: The **-qsmp** option must only be used together with thread-safe compiler invocation modes (those that contain the **_r** suffix).

Parallel regions of program code are executed by multiple threads, possibly running on multiple processors. The number of threads created is determined by environment variables and calls to library functions. Work is distributed among available threads according to scheduling algorithms specified by the environment variables. For any of the methods of parallelization, you can use the XLSMPOPTS environment variable and its suboptions to control thread scheduling; for more information on this environment variable, see "XLSMPOPTS environment variable suboptions for parallel processing" in the *XL C/C++ Compiler Reference*. If you are using OpenMP constructs, you can use the OpenMP environment variables to control thread scheduling; for information on OpenMP environment variables, see "OpenMP environment variables for parallel processing" in the *XL C/C++ Compiler Reference*. For more information on OpenMP built-in functions, see "Built-in functions for parallel processing" in the *XL C/C++ Compiler Reference*.

For a complete discussion on how threads are created and utilized, refer to the *OpenMP Application Program Interface Language Specification*, available at www.openmp.org.

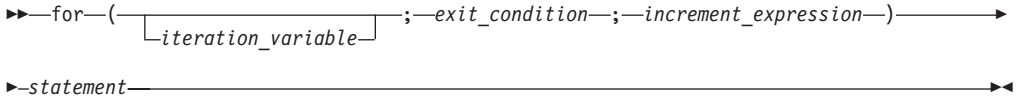
Related information

- "Using shared-memory parallelism (SMP)" on page 42

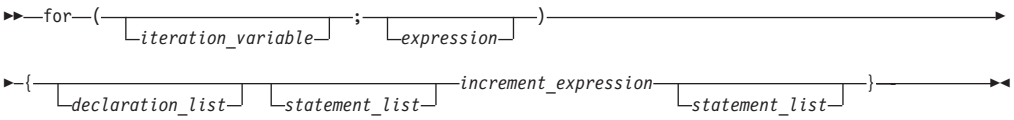
Countable loops

Loops are considered to be *countable* if they take any of the following forms:

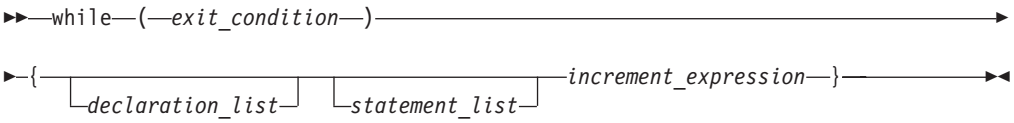
Countable for loop syntax with single statement



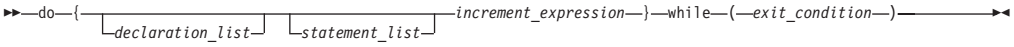
Countable for loop syntax with statement block



Countable while loop syntax



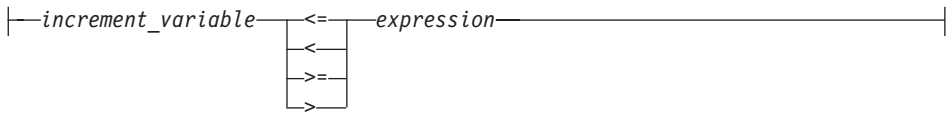
Countable do while loop syntax



The following definitions apply to the above syntax diagrams:

iteration_variable
is a signed integer that has either automatic or register storage class, does not have its address taken, and is not modified anywhere in the loop except in the *increment_expression*.

exit_condition
takes the following form:



where *expression* is a loop-invariant signed integer expression. *expression* cannot reference external or static variables, pointers or pointer expressions, function calls, or variables that have their address taken.

increment_expression
takes any of the following forms:

- *++iteration_variable*
- *--iteration_variable*
- *iteration_variable++*
- *iteration_variable--*
- *iteration_variable += increment*
- *iteration_variable -= increment*
- *iteration_variable = iteration_variable + increment*
- *iteration_variable = increment + iteration_variable*
- *iteration_variable = iteration_variable - increment*

where *increment* is a loop-invariant signed integer expression. The value of the expression is known at run time and is not 0. *increment* cannot reference external or static variables, pointers or pointer expressions, function calls, or variables that have their address taken.

Enabling automatic parallelization

The compiler can automatically locate and where possible parallelize all countable loops in your program code. A loop is considered to be *countable* if it has any of the forms shown in "Countable loops" on page 75, and:

- There is no branching into or out of the loop.
- The increment expression is not within a critical section.

In general, a countable loop is automatically parallelized only if all of the following conditions are met:

- The order in which loop iterations start or end does not affect the results of the program.
- The loop does not contain I/O operations.
- Floating point reductions inside the loop are not affected by round-off error, unless the **-qnostrict** option is in effect.
- The **-qnostrict_induction** compiler option is in effect.
- The **-qsmp=auto** compiler option is in effect.
- The compiler is invoked with a thread-safe compiler mode.

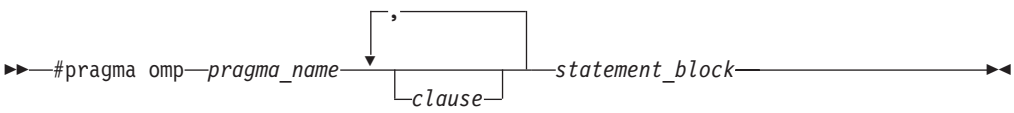
Using OpenMP directives

OpenMP directives exploit shared memory parallelism by defining various types of *parallel regions*. Parallel regions can include both iterative and non-iterative segments of program code.

Pragmas fall into four general categories:

1. Pragmas that let you define parallel regions in which work is done by threads in parallel (**#pragma omp parallel**). Most of the OpenMP directives either statically or dynamically bind to an enclosing parallel region.
2. Pragmas that let you define how work will be distributed or shared across the threads in a parallel region (**#pragma omp section**, **#pragma omp ordered**, **#pragma omp single**).
3. Pragmas that let you control synchronization among threads (**#pragma omp atomic**, **#pragma omp master**, **#pragma omp barrier**, **#pragma omp critical**, **#pragma omp flush)**.
4. Pragmas that let you define the scope of data visibility across threads (**#pragma omp threadprivate**).

OpenMP directive syntax



Pragma directives generally appear immediately before the section of code to which they apply. For example, the following example defines a parallel region in which iterations of a **for** loop can run in parallel:

```
#pragma omp parallel
{
  #pragma omp for
    for (i=0; i<n; i++)
      ...
}
```

This example defines a parallel region in which two or more non-iterative sections of program code can run in parallel:

```
#pragma omp sections
{
  #pragma omp section
     structured_block_1
           ...
  #pragma omp section
     structured_block_2
           ...
      ....
}
```

For a pragma-by-pragma description of the OpenMP directives, refer to "Pragma directives for parallel processing" in the *XL C/C++ Compiler Reference*.

Shared and private variables in a parallel environment

Variables can have either shared or private context in a parallel environment. Variables in shared context are visible to all threads running in associated parallel loops. Variables in private context are hidden from other threads. Each thread has its own private copy of the variable, and modifications made by a thread to its copy are not visible to other threads.

The default context of a variable is determined by the following rules:

- Variables with `static` storage duration are shared.
- Dynamically allocated objects are shared.
- Variables with automatic storage duration are private.
- Variables in heap allocated memory are shared. There can be only one shared heap.
- All variables defined outside a parallel construct become shared when the parallel loop is encountered.
- Loop iteration variables are private within their loops. The value of the iteration variable after the loop is the same as if the loop were run sequentially.
- Memory allocated within a parallel loop by the `alloca` function persists only for the duration of one iteration of that loop, and is private for each thread.

The following code segments show examples of these default rules:

```
int E1;                         /* shared static     */

void main (argvc,...) {         /* argvc is shared   */
   int i;                        /* shared automatic  */

void *p = malloc(...);        /* memory allocated by malloc   */
                                 /* is accessible by all threads */
                                 /* and cannot be privatized     */

#pragma omp parallel firstprivate (p)
   {
     int b;                      /* private automatic  */
     static int s;               /* shared static      */
```

```
     #pragma omp for
     for (i =0;...) {
       b = 1;                    /* b is still private here !    */
       foo (i);                  /* i is private here because it */
                                 /* is an iteration variable     */
      }


#pragma omp parallel
     {
       b = 1;                    /* b is shared here because it  */
                                 /* is another parallel region   */
     }
   }
 }


int E2;                         /*shared static */

void foo (int x) {              /* x is private for the parallel */
                                 /* region it was called from     */

int c;                         /* the same */
 ... }
```

The compiler can privatize some shared variables if it is possible to do so without changing the semantics of the program. For example, if each loop iteration uses a unique value of a shared variable, that variable can be privatized. Privatized shared variables are reported by the **-qinfo=private** option. Use critical sections to synchronize access to all shared variables not listed in this report.

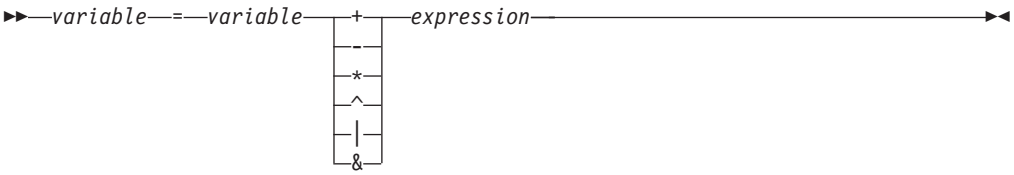
Some OpenMP preprocessor directives let you specify visibility context for selected data variables. A brief summary of data scope attribute clauses are listed below:

Data scope attribute clause	**Description**
private	The **private** clause declares the variables in the list to be private to each thread in a team.
firstprivate	The **firstprivate** clause provides a superset of the functionality provided by the private clause.
lastprivate	The **lastprivate** clause provides a superset of the functionality provided by the private clause.
shared	The **shared** clause shares variables that appear in the list among all the threads in a team. All threads within a team access the same storage area for shared variables.
reduction	The **reduction** clause performs a reduction on the scalar variables that appear in the list, with a specified operator.
default	The **default** clause allows the user to affect the data scope attributes of variables.

For more information, see the OpenMP directive descriptions in ″Pragma directives for parallel processing″ in the *XL C/C++ Compiler Reference* or the *OpenMP Application Program Interface Language Specification*.

Reduction operations in parallelized loops

The compiler can recognize and properly handle most reduction operations in a loop during both automatic and explicit parallelization. In particular, it can handle reduction statements that have either of the following forms:



where:

variable
is an identifier designating an automatic or register variable that does not have its address taken and is not referenced anywhere else in the loop, including all loops that are nested. For example, in the following code, only S in the nested loop is recognized as a reduction:

```
int i,j, S=0;
for (i= 0 ;i < N; i++) {
    S = S+ i;
     for (j=0;j< M; j++) {
        S = S + j;
    }
}
```

expression
is any valid expression.

Recognized reductions are listed by the **-qinfo=reduction** option. OpenMP directives provide you with mechanisms to specify reduction variables explictly.

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director
IBM Canada Ltd. Laboratory
8200 Warden Avenue
Markham, Ontario L6G 1C7
Canada

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1998, 2007. All rights reserved.

Trademarks and service marks

Company, product, or service names identified in the text may be trademarks or service marks of IBM or other companies. Information on the trademarks of International Business Machines Corporation in the United States, other countries, or both is located at http://www.ibm.com/legal/copytrade.shtml.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Industry standards

The following standards are supported:

- The C language is consistent with the International Standard for Information Systems-Programming Language C (ISO/IEC 9899-1990).
- The C language is also consistent with the International Standard for Information Systems-Programming Language C (ISO/IEC 9899-1999 (E)).
- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998).
- The C++ language is also consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:2003 (E)).
- The C and C++ languages are consistent with the OpenMP C and C++ Application Programming Interface Version 2.5.

Index

Special characters

Numerics

A

B

C

D

E

F

H

I

L

M

O

P

R

S

T

V

X

Program Number: 5724-S73

SC23-5890-00