IBM XL C/C++ Advanced Edition V7.0 for Linux

Language Reference

Version 7.0

SC09-7945-01

IBM XL C/C++ Advanced Edition V7.0 for Linux

Language Reference

Version 7.0

SC09-7945-01

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 399.

Second Edition (March, 2005)

This edition applies to Version 7.0.1 of IBM® XL C/C++ Advanced Edition V7.0 for Linux compilers, (product number 5724-K77), and to all subsequent releases and modifications until otherwise indicated in new editions.

IBM welcomes your comments. You can send them by the Internet to the following address:

`compinfo@ca.ibm.com`

Include the title and order number of this book, and the page number or topic related to your comment. Be sure to include your e-mail address if you want a reply.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

- **About This Reference** ix
- Supported language standards ix
- The IBM language extensions x
  - Features related to GNU Cand C++. x
  - Extensions supporting the AltiVec Programming Interface xi
- Document conventions xi
  - Highlighting conventions xi
  - Examples xi
- How to read the syntax diagrams xi
- **Chapter 1. Scope and Linkage** 1
- Scope 2
  - Local Scope 3
  - Function Scope 3
  - Function Prototype Scope 3
  - Global Scope 3
  - Class Scope 4
  - Name Spaces of Identifiers 4
  - Name Hiding 5
- Program Linkage 6
  - Internal Linkage 6
  - External Linkage 7
  - No Linkage 7
- Linkage Specifications — Linking to Non-C++ Programs 8
  - Name Mangling 9
- **Chapter 2. Lexical Elements** 11
- Tokens 11
  - Punctuators 11
    - Alternative Tokens 12
- Source Program Character Set 12
  - Escape Sequences 13
  - The Unicode Standard 14
  - Trigraph Sequences 15
  - Multibyte Characters 16
- Comments 16
- Identifiers 18
  - Reserved Identifiers 18
  - Case Sensitivity and Special Characters in Identifiers 19
  - Predefined Identifiers 19
  - Keywords 20
    - Keywords for Language Extensions 20
    - Alternative Representations of Operators and Punctuators 21
- Literals 21
  - Boolean Literals 21
  - Integer Literals 22
    - Decimal Integer Literals 22
    - Hexadecimal Integer Literals 23
    - Octal Integer Literals 23
  - Floating-Point Literals 23
    - Hexadecimal Floating Constants 25
  - Complex Literals 26
  - Character Literals 26
  - String Literals 27
  - Compound Literals 28
  - Vector Literals 29
- **Chapter 3. Declarations** 31
- Declaration Overview 31
  - Variable Attributes 32
    - The aligned Variable Attribute 32
    - The init_priority Variable Attribute 33
    - The mode Variable Attribute. 33
    - The packed Variable Attribute 34
    - The weak Variable Attribute 34
  - The __align Specifier 34
  - Tentative Definitions 35
- Objects 35
- Storage Class Specifiers 36
  - auto Storage Class Specifier 37
  - extern Storage Class Specifier 38
  - mutable Storage Class Specifier. 39
  - register Storage Class Specifier 39
  - static Storage Class Specifier. 40
  - typedef 42
- Type Specifiers 44
  - Type Names 44
  - Type Attributes 45
    - Type Attribute aligned. 45
    - Type Attribute packed 46
    - Type Attribute transparent_union 46
  - Compatible Types 48
  - Simple Type Specifiers. 49
    - Boolean Variables 49
    - char and wchar_t Type Specifiers 50
      - The wchar_t Type Specifier 51
    - Floating-Point Variables 51
    - Integer Variables. 52
    - void Type 53
  - Compound Types 54
    - Structures 54
      - Declaring and Defining a Structure . . . 55
      - Defining a Structure Variable 57
      - Initializing Structures 57
      - Declaring Structure Types and Variables in the Same Statement. 59
      - Alignment of Structures 60
      - Declaring and Using Bit Fields in Structures 60
    - Unions 63
      - Declaring a Union 64
      - Defining a Union Variable 65
      - Anonymous Unions 65
    - Enumerations. 67
      - Declaring an Enumeration Data Type. . . 67
      - Enumeration Constants 68

Defining Enumeration Variables 68
Defining an Enumeration Type and Enumeration Objects 69
Complex Types 70
Type Qualifiers 71
The const Type Qualifier 73
The volatile Type Qualifier 74
The restrict Type Qualifier 74
The asm Statement 75
Incomplete Types 77
AltiVec Types 78

Chapter 4. Declarators 81

Initializers 82
Pointers 83
Declaring Pointers 84
Assigning Pointers 84
Initializing Pointers 85
Using Pointers 85
Pointer Arithmetic 86
Arrays 87
Declaring Arrays 89
Variable Length Arrays 90
Initializing Arrays 91
Initializing Arrays Using Designated Initializers 93
Function Specifiers 96
References 96
Initializing References 97

Chapter 5. Expressions and Operators 99

Operator Precedence and Associativity 100
Lvalues and Rvalues 103
Primary Expressions 104
Identifier Expressions. 105
Integer Constant Expressions 106
Parenthesized Expressions ( ) 106
C++ Scope Resolution Operator :: 107
Postfix Expressions 108
Function Call Operator ( ) 109
Array Subscripting Operator [ ] 111
Dot Operator 112
Arrow Operator −> 112
The typeid Operator 112
static_cast Operator 114
reinterpret_cast Operator 114
const_cast Operator 116
dynamic_cast Operator 117
Unary Expressions. 119
Increment ++ 120
Decrement −− 120
Unary Plus + ... 121
Unary Minus − 121
Logical Negation ! 121
Bitwise Negation ~ 121
Address & 122
Indirection * 123
alignof Operator 123
sizeof Operator 124
typeof Operator 125
Label Value Operator && 126
C++ new Operator 126
Initializing Objects Created with the new Operator 129
set_new_handler() — Set Behavior for new Failure 129
C++ delete Operator 130
Cast Expressions 131
Cast to a Union Type 132
Binary Expressions 133
Multiplication * 134
Division / 134
Remainder % 135
Addition + ... 135
Subtraction − 135
Bitwise Left and Right Shift << >> 136
Relational < > <= >= 136
Equality == != 138
Bitwise AND & 139
Bitwise Exclusive OR ^ 139
Bitwise Inclusive OR | 140
Logical AND && 140
Logical OR | | 141
C++ Pointer to Member Operators .* −>* . . . 142
Conditional Expressions 142
Type of Conditional C Expressions 143
Type of Conditional C++ Expressions 143
Examples of Conditional Expressions 144
Assignment Expressions 144
Simple Assignment = ... 145
Compound Assignment 145
Comma Expressions 146
C++ throw Expressions 148

Chapter 6. Implicit Type Conversions 149

Integral and Floating-Point Promotions 149
Standard Type Conversions. 150
Lvalue-to-Rvalue Conversions 150
Boolean Conversions 151
Integral Conversions 151
Floating-Point Conversions 152
Pointer Conversions 152
Reference Conversions 153
Pointer-to-Member Conversions 154
Qualification Conversions 154
Function Argument Conversions 154
Other Conversions 155
Arithmetic Conversions 156
The explicit Keyword. 156

Chapter 7. Functions 159

C++ Enhancements to C Functions 159
Function Declarations 160
C++ Function Declarations 162
Multiple Function Declarations 162
Parameter Names in Function Declarations 163
Function Attributes 163
The alias Function Attribute 164
The always_inline Function Attribute . . . 165
The const Function Attribute 165

The constructor and destructor Function Attributes 166
The format Function Attribute 166
The format_arg Function Attribute 167
The noinline Function Attribute 167
The noreturn Function Attribute 168
The pure Function Attribute 168
The section Function Attribute 168
The weak Function Attribute 169
Examples of Function Declarations 169
Function Definitions 170
Ellipsis and void 174
Examples of Function Definitions 174
The main() Function 176
Arguments to main 176
Example of Arguments to main 176
Calling Functions and Passing Arguments 177
Passing Arguments by Value 178
Passing Arguments by Reference 179
Default Arguments in C++ Functions 181
Restrictions on Default Arguments 181
Evaluating Default Arguments 182
Function Return Values 183
Using References as Return Types 184
Allocation and Deallocation Functions 184
Pointers to Functions 185
Inline Functions 186
Nested Functions 187

Chapter 8. Statements 189
Labels 189
Locally Declared Labels 190
Labels as Values 190
Expression Statements 191
Resolving Ambiguous Statements in C++ 191
Block Statement 192
Statement Expressions 193
if Statement 193
switch Statement 195
while Statement 199
do Statement 200
for Statement 200
break Statement 202
continue Statement 203
return Statement 204
Value of a return Expression and Function Value 205
goto Statement 206
Computed goto 207
Null Statement 207

Chapter 9. Preprocessor Directives 209
Preprocessor Overview 209
Preprocessor Directive Format 210
Macro Definition and Expansion (#define) 210
Object-Like Macros 211
Function-Like Macros 211
Variadic Macro Extensions 214
Scope of Macro Names (#undef) 215
# Operator 215
Macro Concatenation with the ## Operator 216
Preprocessor Error Directive (#error) 217
Preprocessor Warning Directive (#warning) 217
File Inclusion (#include) 218
Specialized File Inclusion (#include_next) 219
ISO Standard Predefined Macro Names 219
Conditional Compilation Directives 220
#if, #elif 222
#ifdef 222
#ifndef 223
#else 223
#endif 223
Line Control (#line) 224
Null Directive (#) 225
Pragma Directives (#pragma) 226
Standard Pragmas 226
The _Pragma Operator 226

Chapter 10. Namespaces 229
Defining Namespaces 229
Declaring Namespaces 229
Creating a Namespace Alias 229
Creating an Alias for a Nested Namespace 230
Extending Namespaces 230
Namespaces and Overloading 231
Unnamed Namespaces 231
Namespace Member Definitions 233
Namespaces and Friends 233
Using Directive 234
The using Declaration and Namespaces 234
Explicit Access 235

Chapter 11. Overloading 237
Overloading Functions 237
Restrictions on Overloaded Functions 238
Overloading Operators 239
Overloading Unary Operators 241
Overloading Increment and Decrement 241
Overloading Binary Operators 243
Overloading Assignments 243
Overloading Function Calls 245
Overloading Subscripting 246
Overloading Class Member Access 247
Overload Resolution 247
Implicit Conversion Sequences 249
Resolving Addresses of Overloaded Functions 250

Chapter 12. Classes 253
Declaring Class Types 253
Using Class Objects 254
Classes and Structures 255
Scope of Class Names 256
Incomplete Class Declarations 257
Nested Classes 258
Local Classes 260
Local Type Names 261

Chapter 13. Class Members and Friends 263
Class Member Lists 263
Data Members 264

Member Functions 264
const and volatile Member Functions 266
Virtual Member Functions 266
Special Member Functions 266
Member Scope 267
Pointers to Members 268
The this Pointer 269
Static Members 272
Using the Class Access Operators with Static Members 272
Static Data Members 273
Static Member Functions 275
Member Access 276
Friends 278
Friend Scope 280
Friend Access 283

Chapter 14. Inheritance 285

Derivation 287
Inherited Member Access 290
Protected Members 290
Access Control of Base Class Members 291
The using Declaration and Class Members . . . 292
Overloading Member Functions from Base and Derived Classes 293
Changing the Access of a Class Member . . . 295
Multiple Inheritance 296
Virtual Base Classes 297
Multiple Access 298
Ambiguous Base Classes 299
Virtual Functions 302
Ambiguous Virtual Function Calls 306
Virtual Function Access 307
Abstract Classes 308

Chapter 15. Special Member Functions 311

Constructors and Destructors Overview 311
Constructors. 312
Default Constructors 313
Explicit Initialization with Constructors . . . 314
Initializing Base Classes and Members 316
Construction Order of Derived Class Objects 319
Destructors 320
Free Store 323
Temporary Objects 327
User-Defined Conversions 328
Conversion by Constructor 330
Conversion Functions 331
Copy Constructors 332
Copy Assignment Operators 333

Chapter 16. Templates 337

Template Parameters 338
Type Template Parameters 338
Non-Type Template Parameters 338
Template Template Parameters 339
Default Arguments for Template Parameters . . 339
Template Arguments 340
Template Type Arguments 340
Template Non-Type Arguments 340
Template Template Arguments 342
Class Templates 343
Class Template Declarations and Definitions . . 345
Static Data Members and Templates 345
Member Functions of Class Templates 346
Friends and Templates 346
Function Templates 347
Template Argument Deduction 348
Deducing Type Template Arguments . . . 351
Deducing Non-Type Template Arguments 353
Overloading Function Templates 353
Partial Ordering of Function Templates 354
Template Instantiation 355
Implicit Instantiation 355
Explicit Instantiation 357
Template Specialization 358
Explicit Specialization 358
Definition and Declaration of Explicit Specializations 359
Explicit Specialization and Scope 359
Class Members of Explicit Specializations 360
Explicit Specialization of Function Templates 360
Explicit Specialization of Members of Class Templates 361
Partial Specialization 362
Template Parameter and Argument Lists of Partial Specializations 364
Matching of Class Template Partial Specializations 364
Name Binding and Dependent Names 365
The typename Keyword 366
The Keyword template as Qualifier 366

Chapter 17. Exception Handling . . . 369

The try Keyword 369
Nested Try Blocks 371
catch Blocks 371
Function try block Handlers 372
Arguments of catch Blocks 375
Matching between Exceptions Thrown and Caught 375
Order of Catching 376
The throw Expression 377
Rethrowing an Exception 377
Stack Unwinding 379
Exception Specifications 380
Special Exception Handling Functions 383
unexpected() 383
terminate() 384
set_unexpected() and set_terminate() 385
Example Using the Exception Handling Functions. 386

Appendix A. The IBM C Language Extensions 389

Orthogonal Extensions 389
Existing IBM C Extensions with Individual Option Controls 389
IBM C Extensions: C99 Features as Extensions to C89. 389

IBM C Extensions Related to GNU C 391
Non-Orthogonal Extensions 391
Existing IBM C Extensions with Individual Option Controls 392
IBM C Extensions: C99 Features as Extensions to C89. 392
IBM C Extensions Related to GNU C 392
Extensions for AltiVec Programming Interface Support 392

Appendix B. The IBM C++ Language Extensions 393
Orthogonal Extensions 393
IBM C++ Extensions for Compatibility with C99 393
IBM C++ Extensions Related to GNU C . . . 394
IBM C++ Extensions Related to GNU C++ . . 395
Non-Orthogonal Extensions 395
IBM C++ Extensions for Compatibility with C99 395
IBM C++ Extensions Related to GNU C . . . 395

Appendix C. AltiVec Data Types and Literals 397
Vector Data Types 397
Vector Literals 397

Notices 399
Programming Interface Information 401
Trademarks and Service Marks 401
Industry Standards 401

Index 403

About This Reference

The *C/C++ Language Reference* describes the syntax, semantics, and IBM implementation of the C and C++ programming languages. Syntax and semantics constitute a complete specification of a programming language, but conforming implementations of a language specification can differ because of language extensions. The IBM implementations of C and C++ attest to the organic nature of programming languages, reflecting pragmatic considerations and advances in programming techniques. The language extensions to C and C++ reflect the changing needs of modern programming environments.

The aims of this reference are to provide descriptions of the C and C++ languages and to promote a programming style that emphasizes portability. The expression *Standard C* is a specific term for the current formal definition of the C language, preprocessor, and run-time library. The same naming convention exists for C++. This reference describes an implementation that is consistent with Standard C and Standard C++. The compiler also supports previous language levels.

This reference uses the term *K&R C* to refer to the C language plus the generally accepted extensions produced by Brian Kernighan and Dennis Ritchie that were in use prior to the ISO standardization of C.

The depth of coverage assumes some previous experience with C or another programming language. The intent is to present the syntax and semantics of each language implementation to help you write good programs. The compiler does not enforce certain conventions of programming style, even though they lead to well-ordered programs.

A program that conforms strictly to its language specification will have maximum portability among different environments. In theory, a program that compiles correctly with one standards-conforming compiler will compile and execute correctly under all other conforming compilers, insofar as hardware differences permit. A program that correctly exploits the extensions to the language that are provided by the language implementation can improve the efficiency of its object code.

Supported language standards

The C and C++ languages described in this reference are based on the following standards:

- The C language described in *Programming languages – C* (ISO/IEC 9899:1990), henceforth referred to as *C89*. This was the first ISO C standard.
- The C language described in *Programming languages – C* (ISO/IEC 9899:1999), henceforth referred to as *C99*. This is an update to the C89 standard.
- The C++ language described in *Programming languages – C++* (ISO/IEC 14882:1998), the first formal definition of the language, henceforth referred to as *C++98*.
- The C++ language described in *Programming languages – C++* (ISO/IEC 14882:2003(E)), which is the current meaning of the term *Standard C++*.

The C language described in this reference is consistent with C99 and documents the features supported by XL C/C++. The compiler supports all language features specified in the Standard. Note that the Standard also specifies features in the run-time library. These features may not be supported in the current run-time library and operating environment.

The C++ language described in this reference is consistent with Standard C++ and documents the features supported by XL C/C++.

The IBM language extensions

In addition to the features supported by the various language standards, XL C/C++ contains language extensions that enhance usability and facilitate porting programs to different platforms.

We refer to the following language specifications as ″base language levels″ in order to introduce the notion of an extension to a base.

- Standard C++
- C++98
- C99
- C89

An *orthogonal extension* is a feature that is added on top of a base without altering the behavior of the existing language features. A valid program conforming to a base level will continue to compile and run correctly with such extensions. The program will still be valid, and its behavior will remain unchanged in the presence of the orthogonal extensions. Such an extension is therefore consistent with the corresponding base standard level. Invalid programs may behave differently at execution time and in the diagnostics issued by the compiler.

On the other hand, a *non-orthogonal extension* is one that can change the semantics of existing constructs or can introduce syntax conflicting with the base. A valid program conforming to the base is not guaranteed to compile and run correctly with the non-orthogonal extensions. Because of this, individual compiler options are provided to enable them.

Refer to *XL C/C++ Compiler Reference* for details about `-qlanglvl` and related compiler options for enabling the language extension features.

Features related to GNU Cand C++

Certain language extensions that correspond to GNU C and C++ features are implemented to facilitate portability. These include both orthogonal and non-orthogonal extensions to C89, C99, and C++98. They are controlled by the `-qlanglvl` compiler option, as described in the previous section.

An example of an orthogonal extension related to GNU C is specifying the `noreturn` function attribute in a function declaration and definition. The compiler is informed that the function never returns, which may result in better performance, but any conforming program will not be affected by the feature. The semantics of the `noreturn` function attribute are deemed *orthogonal*.

An example of a non-orthogonal extension is the **inline** keyword. It is non-orthogonal because its current GNU C semantics are different from those of C99.

Extensions supporting the AltiVec Programming Interface

XL C/C++ implements non-orthogonal language features for handling AltiVec vector types. These extensions support language features that exploit the SIMD and parallel processing capabilities of the PowerPC processor, and facilitate the associated optimization techniques.

Document conventions

Highlighting conventions

Bold Identifies names that are predefined by the language, compiler, or operating system, including:

- system commands file and directory names
- programming keywords and library functions
- compiler options, directives, built-in functions, and predefined macros

Italics Identify parameters whose actual names or values are to be supplied by the user. Italics are also used for the first mention of new terms.

`Monospace` Identifies examples of program code, command strings, or user-defined names.

Examples

Examples are intended to be instructional and do not attempt to minimize run time, conserve storage, or check for errors. The examples do not demonstrate all of the possible uses of language constructs. Some examples are only code fragments and will not compile without additional code.

How to read the syntax 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.

  Diagrams of syntactical units other than complete commands, directives, or statements start with the ├── symbol and end with the ──┤ symbol.

  Note: In the following diagrams, `statement` represents a C or C++ command, directive, or statement.

- Required items appear on the horizontal line (the main path).

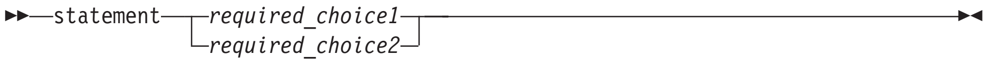
```
►►──statement──required_item──────────────────────────────────►◄
```

- Optional items appear below the main path.

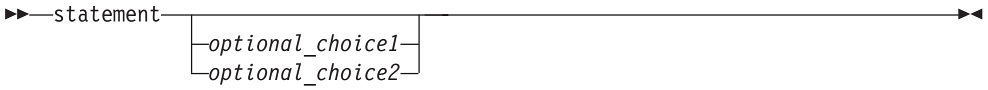
```
►►──statement──┬───────────────┬──────────────────────────────►◄
               └─optional_item─┘
```

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

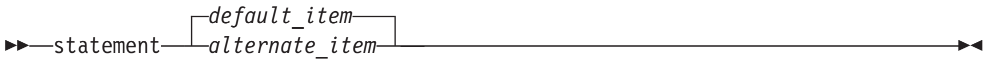
  If you *must* choose one of the items, one item of the stack appears on the main path.



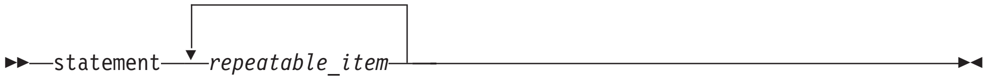
  If choosing one of the items is optional, the entire stack appears below the main path.



  The item that is the default appears above the main path.



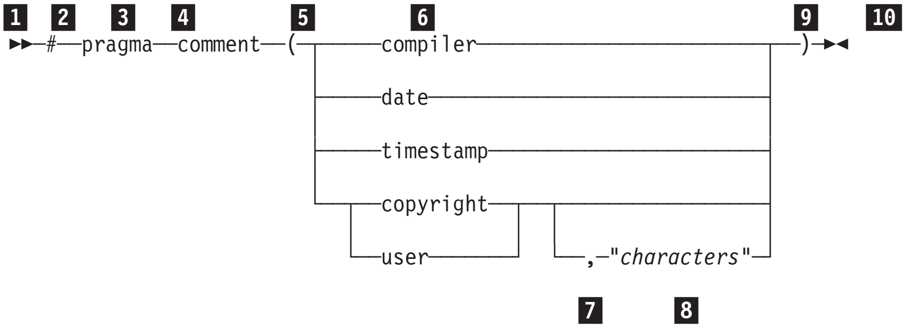
- An arrow returning to the left above the main line indicates an item that can be repeated.



  A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.
- Keywords appear in nonitalic letters and should be entered exactly as shown (for example, `extern`).

  Variables appear in italicized lowercase letters (for example, *identifier*). 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.

The following syntax diagram example shows the syntax for the **#pragma comment** directive. See “Pragma Directives (#pragma)” on page 226 for information on the **#pragma** directive.



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")
```

Chapter 1. Scope and Linkage

Scope is the largest region of program text in which a name can potentially be used without qualification to refer to an entity; that is, the largest region in which the name potentially is valid. Broadly speaking, scope is the general context used to differentiate the meanings of entity names. The rules for scope combined with those for name resolution enable the compiler to determine whether a reference to an identifier is legal at a given point in a file.

The scope of a declaration and the visibility of an identifier are related but distinct concepts. Scope is the mechanism by which it is possible to limit the visibility of declarations in a program. The *visibility* of an identifier is that region of program text from which the object associated with the identifier can be legally accessed. Scope can exceed visibility, but visibility cannot exceed scope. Scope exceeds visibility when a duplicate identifier is used in an inner declarative region, thereby hiding the object declared in the outer declarative region. The original identifier cannot be used to access the first object until the scope of the duplicate identifier (the lifetime of the second object) has ended.

Thus, the scope of an identifier is interrelated with the *storage duration* of the identified object, which is the length of time that an object remains in an identified region of storage. The lifetime of the object is influenced by its storage duration, which in turn was affected by the scope of the object identifier.

Linkage refers to the use or availability of a name across multiple translation units or within a single translation unit. The term *translation unit* refers to a source code file plus all the header and other source files that are included after preprocessing with the `#include` directive, minus any source lines skipped because of conditional preprocessing directives. Linkage allows the correct association of each instance of an identifier with one particular object or function.

Scope and linkage are distinguishable in that scope is for the benefit of the compiler, whereas linkage is for the benefit of the linker. During the translation of a source file to object code, the compiler keeps track of the identifiers that have external linkage and eventually stores them in a table within the object file. The linker is thereby able to determine which names have external linkage, but is unaware of those with internal or no linkage.

Related References

- "Program Linkage" on page 6

Scope

The *scope* of an identifier is the largest region of the program text in which the identifier can potentially be used to refer to its object. In C++, the object being referred to must be unique. However, the name to access the object, the identifier itself, can be reused. The meaning of the identifier depends upon the context in which the identifier is used. Scope is the general context used to distinguish the meanings of names.

The scope of an identifier is possibly noncontiguous. One of the ways that breakage occurs is when the same name is reused to declare a different entity, thereby creating a contained declarative region (inner) and a containing declarative region (outer). Thus, point of declaration is a factor affecting scope. Exploiting the possibility of a noncontiguous scope is the basis for the technique called *information hiding*.

The concept of scope that exists in C was expanded and refined in C++. The following table shows the kinds of scopes and the minor differences in terminology.

Kinds of scope

C	C++
block	local
function	function
function prototype	function prototype
file (global)	global namespace
	namespace
	class

In all declarations, the identifier is in scope before the initializer. The following example demonstrates this:

```
int x;
void f() {

  int x = x;
}
```

The `x` declared in function `f()` has local scope, not global namespace scope.

C++ The remainder of this section pertains to C++ only.

Global scope or *global namespace scope* is the outermost namespace scope of a program, in which objects, functions, types and templates can be defined. A user-defined namespace can be nested within the global scope using namespace definitions, and each user-defined namespace is a different scope, distinct from the global scope.

A function name that is first declared as a friend of a class is in the innermost nonclass scope that encloses the class. A function name that is first declared in an outer namespace will not be used as the friend declaration. For example,

```
int f();

namespace A {
      class X {
            friend f();  // refers to A::f() not to ::f();
      }
      f() { /* definition of f() */ }
}
```

If the friend function is a member of another class, it has the scope of that class. The scope of a class name first declared as a friend of a class is the first nonclass enclosing scope.

The implicit declaration of the class is not visible until another declaration of that same class is seen.

Local Scope

A name has *local scope* or *block scope* if it is declared in a block. A name with local scope can be used in that block and in blocks enclosed within that block, but the name must be declared before it is used. When the block is exited, the names declared in the block are no longer available.

Parameter names for a function have the scope of the outermost block of that function. Also if the function is declared and not defined, these parameter names have function prototype scope.

When one block is nested inside another, the variables from the outer block are usually visible in the nested block. However, if the declaration of a variable in a nested block has the same name as a variable that is declared in an enclosing block, the declaration in the nested block hides the variable that was declared in the enclosing block. The original declaration is restored when program control returns to the outer block. This is called *block visibility.*

Name resolution in a local scope begins in the immediate scope in which the name is used and continues outward with each enclosing scope. The order in which scopes are searched during name resolution causes the phenomenon of information hiding. A declaration in an enclosing scope is hidden by a declaration of the same identifier in a nested scope.

Related References
- “Block Statement” on page 192

Function Scope

The only type of identifier with *function scope* is a label name. A label is implicitly declared by its appearance in the program text and is visible throughout the function that declares it.

A label can be used in a **goto** statement before the actual label is seen.

Related References
- “Labels” on page 189

Function Prototype Scope

In a function declaration (also called a *function prototype*) or in any function declarator—except the declarator of a function definition—parameter names have *function prototype scope*. Function prototype scope terminates at the end of the nearest enclosing function declarator.

Related References
- “Function Declarations” on page 160

Global Scope

▶ C A name has *global scope* if the identifier’s declaration appears outside of any block. A name with global scope and internal linkage is visible from the point where it is declared to the end of the translation unit.

▶ C++ A name has *global namespace scope* if the identifier's declaration appears outside of all blocks, namespaces, and classes. A name with global namespace scope and internal linkage is visible from the point where it is declared to the end of the translation unit.

A name with global (namespace) scope is also accessible for the initialization of global variables. If that name is declared **extern**, it is also visible at link time in all object files being linked.

Related References
- Chapter 10, "Namespaces," on page 229
- "Internal Linkage" on page 6

Class Scope

▶ C++ A name declared within a member function hides a declaration of the same name whose scope extends to or past the end of the member function's class.

When the scope of a declaration extends to or past the end of a class definition, the regions defined by the member definitions of that class are included in the scope of the class. Members defined lexically outside of the class are also in this scope. In addition, the scope of the declaration includes any portion of the declarator following the identifier in the member definitions.

The name of a class member has *class scope* and can only be used in the following cases:
- In a member function of that class
- In a member function of a class derived from that class
- After the `.` (dot) operator applied to an instance of that class
- After the `.` (dot) operator applied to an instance of a class derived from that class, as long as the derived class does not hide the name
- After the `->` (arrow) operator applied to a pointer to an instance of that class
- After the `->` (arrow) operator applied to a pointer to an instance of a class derived from that class, as long as the derived class does not hide the name
- After the `::` (scope resolution) operator applied to the name of a class
- After the `::` (scope resolution) operator applied to a class derived from that class.

Related References
- Chapter 12, "Classes," on page 253
- "Scope of Class Names" on page 256
- "Access Control of Base Class Members" on page 291

Name Spaces of Identifiers

Name spaces are the various syntactic contexts within which an identifier can be used. Within the same context and the same scope, an identifier must uniquely identify an entity. Note that the term *name space* as used here applies to C as well as C++ and does not refer to the C++ namespace language feature. The compiler sets up *name spaces* to distinguish among identifiers referring to different kinds of entities. Identical identifiers in different name spaces do not interfere with each other, even if they are in the same scope.

The same identifier can declare different objects as long as each identifier is unique within its name space. The syntactic context of an identifier within a program lets the compiler resolve its name space without ambiguity.

Within each of the following four name spaces, the identifiers must be unique.

- *Tags* of these types must be unique within a single scope:
  – Enumerations
  – Structures and unions
- *Members* of structures, unions, and classes must be unique within a single structure, union, or class type.
- *Statement labels* have function scope and must be unique within a function.
- All other *ordinary identifiers* must be unique within a single scope:
  – C function names (C++ function names can be overloaded)
  – Variable names
  – Names of function parameters
  – Enumeration constants
  – `typedef` names.

You can redefine identifiers in the same name space but within enclosed program blocks.

Structure tags, structure members, variable names, and statement labels are in four different name spaces. No name conflict occurs among the items named `student` in the following example:

```
int get_item()
{
   struct student     /* structure tag                  */
   {
      char name[20];  /* this structure member may not be named student      */
      int section;
      int id;
   } sam;             /* this structure variable should not be named student */

   goto student;
   student:;          /* null statement label            */
   return 0;

   student fred;      /* legal struct declaration in C++ */
}
```

The compiler interprets each occurrence of `student` by its context in the program. For example, when `student` appears after the keyword `struct`, it is a structure tag. The name `student` may not be used for a structure member of `struct student`. When `student` appears after the `goto` statement, the compiler passes control to the null statement label. In other contexts, the identifier `student` refers to the structure variable.

Name Hiding

▶ C++ If a class name or enumeration name is in scope and not hidden it is *visible*. A class name or enumeration name can be hidden by an explicit declaration of that same name — as an object, function, or enumerator — in a nested declarative region or derived class. The class name or enumeration name is hidden wherever the object, function, or enumerator name is visible. This process is referred to as *name hiding*.

In a member function definition, the declaration of a local name hides the declaration of a member of the class with the same name. The declaration of a member in a derived class hides the declaration of a member of a base class of the same name.

Suppose a name x is a member of namespace A, and suppose that the members of namespace A are visible in a namespace B because of a using declaration. A declaration of an object named x in namespace B will hide A::x. The following example demonstrates this:

```
#include <iostream>
#include <typeinfo>
using namespace std;

namespace A {
  char x;
};

namespace B {
  using namespace A;
  int x;
};

int main() {
  cout << typeid(B::x).name() << endl;
}
```

The following is the output of the above example:

```
int
```

The declaration of the integer x in namespace B hides the character x introduced by the using declaration.

Related References

- Chapter 12, "Classes," on page 253
- "Member Functions" on page 264
- "Member Scope" on page 267
- Chapter 10, "Namespaces," on page 229

Program Linkage

Linkage determines whether identifiers that have identical names refer to the same object, function, or other entity, even if those identifiers appear in different translation units. The linkage of an identifier depends on how it was declared. There are three types of linkages: external, internal, and no linkage.

- Identifiers with *external linkage* can be seen (and refered to) in other translation units.
- Identifiers with *internal linkage* can only be seen within the translation unit.
- Identifiers with *no linkage* can only be seen in the scope in which they are defined.

Linkage does not affect scoping, and normal name lookup considerations apply.

▶ C++ You can also have linkage between C++ and non-C++ code fragments, which is called *language linkage*. Language linkage enables the close relationship between C++ and C by allowing C++ code to link with that written in C. All identifiers have a language linkage, which by default is C++. Language linkage must be consistent across translation units, and non-C++ language linkage implies that the identifier has external linkage.

Internal Linkage

The following kinds of identifiers have internal linkage:

- Objects, references, or functions explicitly declared **static**.

- Objects or references declared in namespace scope (or global scope in C) with the specifier **const** and neither explicitly declared **extern**, nor previously declared to have external linkage.
- Data members of an anonymous union.
- C++ Function templates explicitly declared **static**.
- C++ Identifiers declared in the unnamed namespace.

A function declared inside a block will usually have external linkage. An object declared inside a block will usually have external linkage if it is specified **extern**. If a variable that has **static** storage is defined outside a function, the variable has internal linkage and is available from the point where it is defined to the end of the current translation unit.

If the declaration of an identifier has the keyword **extern** and if a previous declaration of the identifier is visible at namespace or global scope, the identifier has the same linkage as the first declaration.

External Linkage

C In global scope, identifiers for the following kinds of entities declared without the **static** storage class specifier have external linkage:

- An object.
- A function.

If an identifier in C is declared with the **extern** keyword and if a previous declaration of an object or function with the same identifier is visible, the identifier has the same linkage as the first declaration. For example, a variable or function that is first declared with the keyword **static** and later declared with the keyword **extern** has internal linkage. However, a variable or function that has no linkage and was later declared with a linkage specifier will have the linkage that was expressly specified.

C++ In namespace scope, the identifiers for the following kinds of entities have external linkage:

- A reference or an object that does not have internal linkage.
- A function that does not have internal linkage.
- A named class or enumeration.
- An unnamed class or enumeration defined in a typedef declaration.
- An enumerator of an enumeration that has external linkage.
- A template, unless it is a function template with internal linkage.
- A namespace, unless it is declared in an unnamed namespace.

If the identifier for a class has external linkage, then, in the implementation of that class, the identifiers for the following will also have external linkage:

- A member function.
- A static data member.
- A class of class scope.
- An enumeration of class scope.

No Linkage

The following kinds of identifiers have no linkage:

- Names that have neither external or internal linkage
- Names declared in local scopes (with exceptions like certain entities declared with the **extern** keyword)

- Identifiers that do not represent an object or a function, including labels, enumerators, **typedef** names that refer to entities with no linkage, type names, function parameters, and template names

You cannot use a name with no linkage to declare an entity with linkage. For example, you cannot use the name of a class or enumeration or a **typedef** name referring to an entity with no linkage to declare an entity with linkage. The following example demonstrates this:

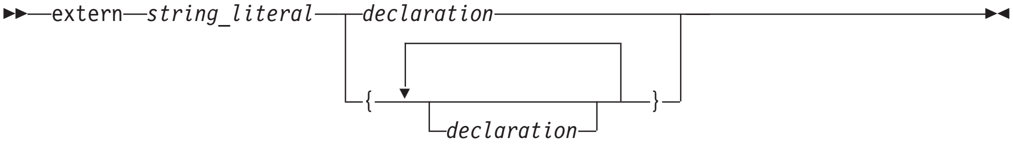
```
int main() {
  struct A { };
//  extern A a1;
  typedef A myA;
//  extern myA a2;
}
```

The compiler will not allow the declaration of `a1` with external linkage. Class `A` has no linkage. The compiler will not allow the declaration of `a2` with external linkage. The **typedef** name `a2` has no linkage because `A` has no linkage.

Linkage Specifications — Linking to Non-C++ Programs

C++ Linkage between C++ and non-C++ code fragments is called *language linkage*. All function types, function names, and variable names have a language linkage, which by default is C++.

You can link C++ object modules to object modules produced using other source languages such as C by using a *linkage specification*. The syntax is:



The *string_literal* is used to specify the linkage associated with a particular function. String literals used in linkage specifications should be considered as case-sensitive. All platforms support the following values for *string_literal*

`"C++"`	Unless otherwise specified, objects and functions have this default linkage specification.
`"C"`	Indicates linkage to a C procedure

Calling shared libraries that were written before C++ needed to be taken into account requires the `#include` directive to be within an `extern "C" {}` declaration.

```
extern "C" {
#include "shared.h"
}
```

The following example shows a C printing function that is called from C++.

```
//  in C++ program
extern "C" int displayfoo(const char *);
int main() {
    return displayfoo("hello");
}

/*  in C program     */
#include <stdio.h>
```

```
extern int displayfoo(const char * str) {
    while (*str) {
       putchar(*str);
       putchar(' ');
       ++str;
    }
    putchar('\n');
}
```

Name Mangling

C++ Name mangling is the encoding of function and variable names into unique names so that linkers can separate common names in the language. Type names may also be mangled. The compiler generates function names with an encoding of the types of the function arguments when the module is compiled. Name mangling is commonly used to facilitate the overloading feature and visibility within different scopes. Name mangling also applies to variable names. If a variable is in a namespace, the name of the namespace is mangled into the variable name so that the same variable name can exist in more than one namespace. The C++ compiler also mangles C variable names to identify the namespace in which the C variable resides.

The scheme for producing a mangled name differs with the object model used to compile the source code: the mangled name of an object of a class compiled using one object model will be different from that of an object of the same class compiled using a different object model. The object model is controlled by compiler option or by pragma.

Name mangling is not desirable when linking C modules with libraries or object files compiled with a C++ compiler. To prevent the C++ compiler from mangling the name of a function, you can apply the `extern "C"` linkage specifier to the declaration or declarations, as shown in the following example:

```
extern "C" {
   int f1(int);
   int f2(int);
   int f3(int);
};
```

This declaration tells the compiler that references to the functions `f1`, `f2`, and `f3` should not be mangled.

The `extern "C"` linkage specifier can also be used to prevent mangling of functions that are defined in C++ so that they can be called from C. For example,

```
extern "C" {
   void p(int){
      /* not mangled */
   }
};
```

In multiple levels of nested **extern** declarations, the innermost **extern** specification prevails.

```
extern "C" {
      extern "C++" {
            void func();
      }
}
```

In this example, `func()` has C++ linkage.

Chapter 2. Lexical Elements

A *lexical element* refers to a character or groupings of characters that may legally appear in a source file. This section contains discussions of the basic lexical elements and conventions of the C and C++ programming languages: tokens, character sets, comments, identifiers, and literals.

Tokens

Source code is treated during preprocessing and compilation as a sequence of *tokens*. A token is the smallest independent unit of meaning in a program, as defined by the compiler. There are five different types of tokens:

- Identifiers
- Keywords
- Literals
- Operators
- Punctuators

Adjacent identifiers, keywords, and literals must be separated with white space. Other tokens should be separated by white space to make the source code more readable. White space includes blanks, horizontal and vertical tabs, new lines, form feeds, and comments.

Punctuators

A *punctuator* is a token that has syntactic and semantic meaning to the compiler, but the exact significance depends on the context. A punctuator can also be a token that is used in the syntax of the preprocessor. At the C89 language level, a punctuator does not cause an action. For example, a comma is a punctuator in an argument list or in an initializer list, but is an operator when used within a parenthesized expression.

At the C89 language level, a punctuator can be a character that separates tokens, such as:

```
[   ]     (   )     {   }     ,          :          ;
```

or any of the following:

```
*          =          ...        #
```

C89 restricts the use of the number sign # to preprocessor directives only.

At the C99 language level, the number of legal tokens for a punctuator or preprocessing token increases to include the C operators. A punctuator that specifies an operation to be performed is known as an *operator*. This distinction between a punctuator and operator is also used by C++. In addition to the C89 punctuators, C99 defines the following tokens as punctuators, operators, or preprocessing tokens:

```
.          ->         ++         --         ##
&          +          -          ~          !
/          %          <<         >>         !=
<          >          <=         >=         ==
^              |              &&             ||             ?
*=             /=             %=             +=             -=
<<=            >>=            &=             ^=             |=
<:             :>             <%             %>             %:             %:%:
```

C++ In addition to the C99 preprocessing tokens, operators, and punctuators, C++ allows the following tokens as punctuators:

```
::             .*             ->*            new            delete
and            and_eq         bitand         bitor          comp
not            not_eq         or             or_eq          xor            xor_eq
```

Alternative Tokens

C and C++ provide alternative representations for some operators and punctuators. The following table lists the operators and punctuators and their alternative representation:

Operator or Punctuator	Alternative Representation
{	<%
}	%>
[	<:
]	:>
#	%:
##	%:%:
&&	and
|	bitor
||	or
^	xor
~	compl
&	bitand
&=	and_eq
|=	or_eq
^=	xor_eq
!	not
!=	not_eq

Source Program Character Set

The following lists the basic *source character set* that must be available at both compile and run time:

- The uppercase and lowercase letters of the English alphabet

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

- The decimal digits 0 through 9

```
0 1 2 3 4 5 6 7 8 9
```

- The following graphic characters:

```
! " # % & ' ( ) * + , - . / :
; < = > ? [ \ ] _ { } ~
```

  - The caret (^) character in ASCII (bitwise exclusive OR symbol).
  - The split vertical bar (¦) character in ASCII.
- The space character

- The control characters representing new-line, horizontal tab, vertical tab, and form feed, and end of string (`NULL` character)

Depending on the implementation and compiler option, other specialized identifiers, such as the dollar sign ($) or characters in national character sets, may be allowed to appear in an identifier.

Escape Sequences

You can represent any member of the execution character set by an *escape sequence*. They are primarily used to put nonprintable characters in character and string literals. For example, you can use escape sequences to put such characters as tab, carriage return, and backspace into an output stream.

```
►►──\──┬─escape_sequence_character─┬──────────────────────────────────────────►◄
       ├─x──hexadecimal_digits─────┤
       └─octal_digits──────────────┘
```

An escape sequence contains a backslash (\) symbol followed by one of the escape sequence characters or an octal or hexadecimal number. A hexadecimal escape sequence contains an `x` followed by one or more hexadecimal digits (0-9, A-F, a-f). An octal escape sequence uses up to three octal digits (0-7). The value of the hexadecimal or octal number specifies the value of the desired character or wide character.

Note: The line continuation sequence (\ followed by a new-line character) is not an escape sequence. It is used in character strings to indicate that the current line of source code continues on the next line.

The escape sequences and the characters they represent are:

Escape Sequence	Character Represented
`\a`	Alert (bell, alarm)
`\b`	Backspace
`\f`	Form feed (new page)
`\n`	New-line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\'`	Single quotation mark
`\"`	Double quotation mark
`\?`	Question mark
`\\`	Backslash

The value of an escape sequence represents the member of the character set used at run time. Escape sequences are translated during preprocessing. For example, on a system using the ASCII character codes, the value of the escape sequence `\x56` is the letter `V`. On a system using EBCDIC character codes, the value of the escape sequence `\xE5` is the letter `V`.

Use escape sequences only in character constants or in string literals. An error message is issued if an escape sequence is not recognized.

In string and character sequences, when you want the backslash to represent itself (rather than the beginning of an escape sequence), you must use a \\ backslash escape sequence. For example:

```
cout << "The escape sequence \\n." << endl;
```

This statement results in the following output:

```
The escape sequence \n.
```

The Unicode Standard

The *Unicode Standard* is the specification of an encoding scheme for written characters and text. It is a universal standard that enables consistent encoding of multilingual text and allows text data to be interchanged internationally without conflict. The ISO standards for C and C++ refer to *ISO/IEC 10646–1:2000, Information Technology—Universal Multiple-Octet Coded Character Set (UCS)*. (The term *octet* is used by ISO to refer to a byte.) The ISO/IEC 10646 standard is more restrictive than the Unicode Standard in the number of encoding forms: a character set that conforms to ISO/IEC 10646 is also conformant to the Unicode Standard.

The Unicode Standard specifies a unique numeric value and name for each character and defines three encoding forms for the bit representation of the numeric value. The name/value pair creates an identity for a character. The hexadecimal value representing a character is called a *code point*. The specification also describes overall character properties, such as case, directionality, alphabetic properties, and other semantic information for each character. Modeled on ASCII, the Unicode Standard treats alphabetic characters, ideographic characters, and symbols, and allows implementation-defined character codes in reserved code point ranges. The encoding scheme of the Unicode Standard is therefore sufficiently flexible to handle all known character encoding requirements, including coverage of historical scripts from any country in the world.

C99 and C++ allow the universal character name construct defined in ISO/IEC 10646 to represent characters outside the basic source character set. Both languages permit universal character names in identifiers, character constants, and string literals. In C++, this language feature is independent of the language level specified at compile time.

The following table shows the generic universal character name construct and how it corresponds to the ISO/IEC 10646 short name.

Universal character name	ISO/IEC 10646 short name
`\UNNNNNNNN`	`NNNNNNNN`
`\uNNNN`	`0000NNNN`
where `N` *is a hexadecimal digit*	

C99 and C++ disallow the hexadecimal values representing characters in the basic character set (base source code set) and the code points reserved by ISO/IEC 10646 for control characters. The following characters are also disallowed:

- Any character whose short identifier is less than `00A0`. The exceptions are `0024` ($), `0040` (@), or `0060` (`).
- Any character whose short identifier is in the code point range `D800` through `DFFF` inclusive.

XL C/C++ implements the data types **uint_least16_t** and **uint_least32_t** to process UTF-16 and UTF-32 characters in C and C++ in conformance with the Unicode

Standard. The data types, also referred to as *u-literals* and *U-literals*, respectively, are the string literals required by the Unicode Standard to specify a UTF-16 or UTF-32 character, and were approved by the C Standards Committee. Previously, a UTF-16 character was represented by an **unsigned short**, and a UTF-32 character, by an **unsigned int**.

The support for *u-literals* and *U-literals* is similar to that for wide character literals.

`u"s-char-sequence"`
Denotes an array of **uint_least16_t**. The corresponding character literal is denoted by

```
U'c-char-sequence'
```

`U"s-char-sequence"`
Denotes an array of **uint_least32_t**. The corresponding character literal is denoted by

```
U'c-char-sequence'
```

For example,

```
uint_least16_t  msg[] = u"ucs characters \u1234 and \U81801234 ";
```

String concatenation

The u-literals and U-literals follow the same concatenation rule as wide character literals: the normal character string is widened if they are present. The following shows the allowed combinations. All other combinations are invalid.

Combination	Result
`u"a" u"b"`	`u"ab"`
`u"a" "b"`	`u"ab"`
`"a" u"b"`	`u"ab"`
`U"a" U"b"`	`U"ab"`
`U"a" "b"`	`U"ab"`
`"a" U"b"`	`U"ab"`

Multiple concatentations are allowed, with these rules applied recursively.

Trigraph Sequences

Some characters from the C and C++ character set are not available in all environments. You can enter these characters into a C or C++ source program using a sequence of three characters called a *trigraph*. The trigraph sequences are:

Trigraph	Single character	Description
??=	#	pound sign
??(	[	left bracket
??)	]	right bracket
??<	{	left brace
??>	}	right brace
??/	\	backslash
??’	^	caret
??!	|	vertical bar
??-	~	tilde

The preprocessor replaces trigraph sequences with the corresponding single-character representation.

Multibyte Characters

A *multibyte character* is a character whose bit representation fits into one or more bytes and is a member of the *extended character set*. The extended character set is a superset of the basic character set. The term *wide character* is a character whose bit representation accommodates an object of type `wchar_t`, capable of representing any character in the current locale.

Related References
- "char and wchar_t Type Specifiers" on page 50

Comments

A *comment* is text replaced during preprocessing by a single space character; the compiler therefore ignores all comments.

There are two kinds of comments:
- The `/*` (slash, asterisk) characters, followed by any sequence of characters (including new lines), followed by the `*/` characters. This kind of comment is commonly called a *C-style comment*.
- The `//` (two slashes) characters followed by any sequence of characters. A new line not immediately preceded by a backslash terminates this form of comment. This kind of comment is commonly called a *single-line comment* or a *C++ comment*. A C++ comment can span more than one physical source line if it is joined into one logical source line with line-continuation (\) characters. The backslash character can also be represented by a trigraph.

You can put comments anywhere the language allows white space. You cannot nest C-style comments inside other C-style comments. Each comment ends at the first occurrence of `*/`.

Multibyte characters can also be included within a comment.

Note: The `/*` or `*/` characters found in a character constant or string literal do not start or end comments.

In the following program, the second `printf()` is a comment:

```
#include <stdio.h>

int main(void)
{
   printf("This program has a comment.\n");
   /* printf("This is a comment line and will not print.\n"); */
return 0;
}
```

Because the second `printf()` is equivalent to a space, the output of this program is:

```
This program has a comment.
```

Because the comment delimiters are inside a string literal, `printf()` in the following program is not a comment.

```
#include <stdio.h>

    int main(void)
    {
       printf("This program does not have \
    /* NOT A COMMENT */ a comment.\n");
    return 0;
    }
```

The output of the program is:

```
This program does not have
/* NOT A COMMENT */ a comment.
```

In the following example, the comments are highlighted:

```
/* A program with nested comments. */

    #include <stdio.h>

    int main(void)
    {
       test_function();
       return 0;
    }

    int test_function(void)
    {
       int number;
       char letter;
    /*
    number = 55;
    letter = 'A';
    /* number = 44; */
    */
    return 999;
    }
```

In test_function, the compiler reads the first /* through to the first */. The second */ causes an error. To avoid commenting over comments already in the source code, you should use conditional compilation preprocessor directives to cause the compiler to bypass sections of a program. For example, instead of commenting out the above statements, change the source code in the following way:

```
    /* A program with conditional compilation to avoid nested comments.
*/
    #define TEST_FUNCTION 0
    #include <stdio.h>

    int main(void)
    {
       test_function();
       return 0;
    }

    int test_function(void)
    {
        int number;
        char letter;
     #if TEST_FUNCTION
       number = 55;
       letter = 'A';
       /*number = 44;*/
     #endif  /*TEST_FUNCTION */
    }
```

You can nest single line comments within C-style comments. For example, the following program will not output anything:

```
#include <stdio.h>

   int main(void)
   {
      /*
      printf("This line will not print.\n");
      // This is a single line comment
      // This is another single line comment
      printf("This line will also not print.\n");
      */
      return 0;
   }
```

Related References

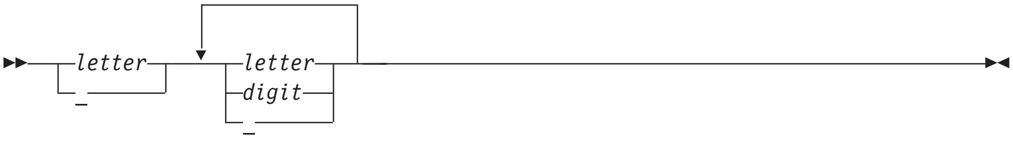
- "Trigraph Sequences" on page 15

Identifiers

Identifiers provide names for the following language elements:

- Functions
- Objects
- Labels
- Function parameters
- Macros and macro parameters
- Typedefs
- Enumerated types and enumerators
- Struct and union names
- C++ Classes and class members
- C++ Templates
- C++ Template parameters
- C++ Namespaces

An identifier consists of an arbitrary number of letters, digits, or the underscore character in the form:



The universal character names for letters and digits outside of the basic source character set are allowed in C++ and at the C99 language level.

Reserved Identifiers

Identifiers with two initial underscores or an initial underscore followed by an uppercase letter are reserved globally for use by the compiler.

C Identifiers that begin with an underscore are reserved as identifiers with file scope in both the ordinary and tag name spaces.

C++ C++ extends the C reservations to include more identifiers in a larger name space. Any name that contains double underscores anywhere is reserved. Any identifier that begins with an underscore is reserved in the global namespace.

Case Sensitivity and Special Characters in Identifiers

The compiler distinguishes between uppercase and lowercase letters in identifiers. For example, `PROFIT` and `profit` represent different identifiers.

Avoid creating identifiers that begin with an underscore (_) for function names and variable names.

The first character in an identifier must be a letter. The _ (underscore) character is considered a letter; however, identifiers beginning with an underscore are reserved by the compiler for identifiers at global namespace scope.

Identifiers that contain two consecutive underscores or begin with an underscore followed by a capital letter are reserved in all contexts.

The dollar sign can appear in identifier names when compiled using the `-qdollar` compiler option or at one of the extended language levels that encompasses this option.

You should always include the appropriate headers when using standard library functions.

Although the names of system calls and library functions are not reserved words if you do not include the appropriate headers, avoid using them as identifiers. Duplication of a predefined name can lead to confusion for the maintainers of your code and can cause errors at link time or run time. If you include a library in a program, be aware of the function names in that library to avoid name duplications. You should always include the appropriate headers when using standard library functions.

Predefined Identifiers

The predefined identifier `__func__` makes the function name available for use within the function. Immediately following the opening brace of each function definition, `__func__` is implicitly declared by the compiler. The resulting behavior is as if the following declaration had been made:

```
static const char __func__[] = "function-name";
```

where *`function-name`* is the name of the lexically-enclosing function. The function name is not mangled.

When this identifier is used with the `assert` macro, the macro adds the name of the enclosing function on the standard error stream.

▶ C++ C++ supports the `__func__` predefined identifier as a language extension for compatibility with C99.

The function name is qualified with the enclosing class name or function name. For example, `foo` is a member function of class `C`. The predefined identifier of `foo` is `C::foo`. If `foo` is defined within the body of `main`, the predefined identifier of foo is `main::C::foo`.

The names of template functions or member functions reflect the instantiated type. For example, the predefined identifier for the template function `foo` instantiated with **int**,

```
template<classT> void foo()
```

is `foo<int>`.

Keywords

Keywords are identifiers reserved by the language for special use. Although you can use them for preprocessor macro names, it is poor programming style. Only the exact spelling of keywords is reserved. For example, `auto` is reserved but `AUTO` is not. The following lists the keywords common to both the C and C++ languages:

- auto
- break
- case
- char
- const
- continue
- default
- do
- double
- else
- enum
- extern
- float
- for
- goto
- if
- inline
- int
- long
- register
- return
- short
- signed
- sizeof
- static
- struct
- switch
- typedef
- union
- unsigned
- void
- volatile
- while

C The C language also reserves the following keywords:

- restrict
- _Bool
- _Complex
- _Imaginary
- uint_least16_t
- uint_least32_t

C++ The C++ language also reserves the following keywords:

- asm
- bool
- catch
- class
- const_cast
- delete
- dynamic_cast
- explicit
- export
- false
- friend
- mutable
- namespace
- new
- operator
- private
- protected
- public
- reinterpret_cast
- static_cast
- template
- this
- throw
- true
- try
- typeid
- typename
- using
- virtual
- wchar_t

Keywords for Language Extensions

Linux In addition to standard language keywords, XL C/C++ reserves identifiers for language extensions, ease of porting applications developed with the GNU C compiler, and for future use. The following keywords are reserved for use in language extensions:

- pixel
- typeof
- vector
- __align
- __asm
- __pixel
- __restrict
- __vector
- __alignof__
- __asm__
- __attribute__
- __complex__
- __const__
- __extension__
- __imag__
- __inline__
- __label__
- __real__
- __signed__
- __typeof__
- __volatile__

The tokens **vector**, **pixel**, and **bool** are recognized as keywords only when used in a vector declaration context and when the AltiVec language extensions are enabled.

C++ The IBM implementation of C++ reserves the following keywords as language extensions for compatibility with C99.

restrict	**_Complex**	**uint_least16_t**
__restrict__	**_Imaginary**	**uint_least32_t**

Alternative Representations of Operators and Punctuators

In addition to the reserved language keywords, the following alternative representations of operators and punctuators are also reserved in C and C++:

and	**bitor**	**not_eq**	**xor**
and_eq	**compl**	**or**	**xor_eq**
bitand	**not**	**or_eq**	

Literals

The term *literal constant* or *literal* refers to a value that occurs in a program and cannot be changed. The C language uses the term *constant* in place of the noun *literal*. The adjective *literal* adds to the concept of a constant the notion that we can speak of it only in terms of its value. A literal constant is nonaddressable, which means that its value is stored somewhere in memory, but we have no means of accessing that address.

Every *literal* has a value and a data type. The value of any literal does not change while the program runs and must be in the range of representable values for its type. The following are the available types of literals:
- Boolean
- Integer
- Character
- Floating-point
- String
- Compound literal
- Vector literal

C C99 adds the compound literal as a postfix expression. The language feature provides a way to specify constants of aggregate or union type.

Linux Mac OS X The Pascal string form of a string literal is also accepted.

Boolean Literals

C The C language does not define any Boolean literals, but instead uses the integer values 0 and 1 to represent boolean values. The value zero represents ″false″ and all nonzero values represent ″true.″

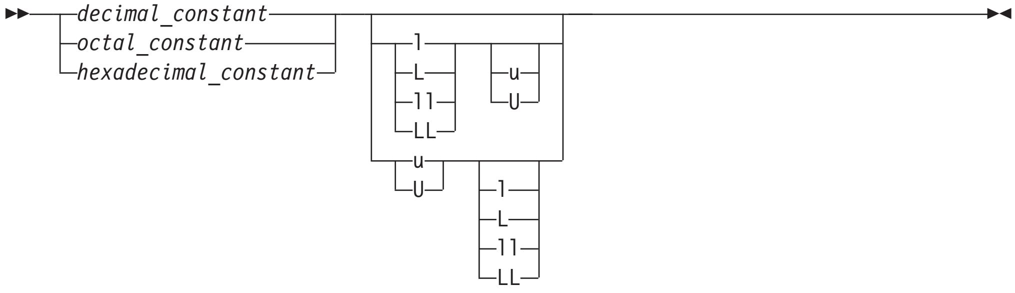
C defines ″true″ and ″false″ as macros in the header file `<stdbool.h>`. When these macros are defined, the macro `__bool_true_false_are_defined` is expanded to the integer constant 1.

C++ There are only two boolean literals: **true** and **false**. These literals have type **bool** and are not lvalues.

Related References
- “Boolean Variables” on page 49
- “Lvalues and Rvalues” on page 103

Integer Literals

Integer literals can represent decimal, octal, or hexadecimal values. They are numbers that do not have a decimal point or an exponential part. However, an integer literal may have a prefix that specifies its base, or a suffix that specifies its type.



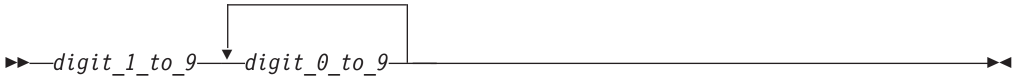
The data type of an integer literal is determined by its form, value, and suffix. The following table lists the integer literals and shows the possible data types. The smallest data type that can represent the constant value is used to store the constant.

Integer Literal	Possible Data Types
unsuffixed decimal	**int, long int, unsigned long int, long long int**
unsuffixed octal	**int, unsigned int, long int, unsigned long int, long long int, unsigned long long int**
unsuffixed hexadecimal	**int, unsigned int, long int, unsigned long int, long long int, unsigned long long int**
decimal, octal, or hexadecimal suffixed by **u** or **U**	**unsigned int, unsigned long int, unsigned long long int**
decimal suffixed by **l** or **L**	**long int, long long int**
octal or hexadecimal suffixed by **l** or **L**	**long int, unsigned long int, long long int, unsigned long long int**
decimal, octal, or hexadecimal suffixed by both **u** or **U**, and **l** or **L**	**unsigned long int**, **unsigned long long int**
decimal suffixed by **ll** or **LL**	**long long int**
octal or hexadecimal suffixed by **ll** or **LL**	**long long int, unsigned long long int**
decimal, octal, or hexadecimal suffixed by both **u** or **U**, and **ll** or **LL**	**unsigned long long int**

A plus (+) or minus (-) symbol can precede an integer literal. The operator is treated as a unary operator rather than as part of the literal.

Decimal Integer Literals

A *decimal integer literal* contains any of the digits 0 through 9. The first digit cannot be 0.



Integer literals beginning with the digit 0 are interpreted as an octal integer literal rather than as a decimal integer literal.

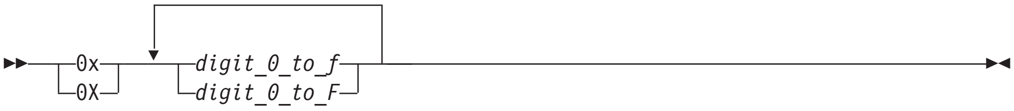
The following are examples of decimal literals:

```
485976
-433132211
+20
5
```

A plus (+) or minus (-) symbol can precede the decimal integer literal. The operator is treated as a unary operator rather than as part of the literal.

Hexadecimal Integer Literals

A *hexadecimal integer literal* begins with the 0 digit followed by either an x or X, followed by any combination of the digits 0 through 9 and the letters a through f or A through F. The letters A (or a) through F (or f) represent the values 10 through 15, respectively.

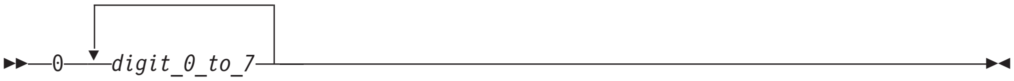


The following are examples of hexadecimal integer literals:

```
0x3b24
0XF96
0x21
0x3AA
0X29b
0X4bD
```

Octal Integer Literals

An *octal integer literal* begins with the digit 0 and contains any of the digits 0 through 7.



The following are examples of octal integer literals:

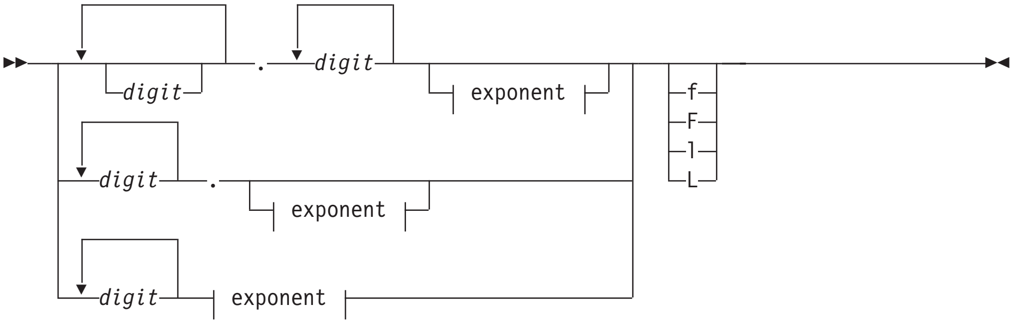
```
0
0125
034673
03245
```

Floating-Point Literals

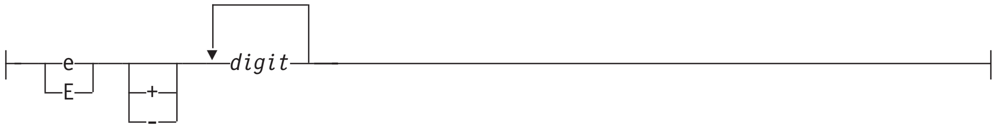
A *floating-point literal* consists of the following:

- An integral part
- A decimal point
- A fractional part
- An exponent part
- An optional suffix

Both the integral and fractional parts are made up of decimal digits. You can omit either the integral part or the fractional part, but not both. You can omit either the decimal point or the exponent part, but not both.



Exponent:



The magnitude range of **float** is approximately 1.2e-38 to 3.4e38. The magnitude range of **double** or **long double** is approximately 2.2e-308 to 1.8e308. If a floating-point constant is too large or too small, the result is undefined by the language.

The suffix **f** or **F** indicates a type of **float**, and the suffix **l** or **L** indicates a type of **long double**. If a suffix is not specified, the floating-point constant has a type **double**.

A plus (**+**) or minus (**-**) symbol can precede a floating-point literal. However, it is not part of the literal; it is interpreted as a unary operator.

The following are examples of floating-point literals:

Floating-Point Constant	Value
`5.3876e4`	`53,876`
`4e-11`	`0.00000000004`
`1e+5`	`100000`
`7.321E-3`	`0.007321`
`3.2E+4`	`32000`
`0.5e-6`	`0.0000005`
`0.45`	`0.45`
`6.e10`	`60000000000`

▶ C When you use the `printf` function to display a floating-point constant value, make certain that the `printf` conversion code modifiers that you specify are large enough for the floating-point constant value.

Related References

- "Floating-Point Variables" on page 51

- "Unary Expressions" on page 119

Hexadecimal Floating Constants

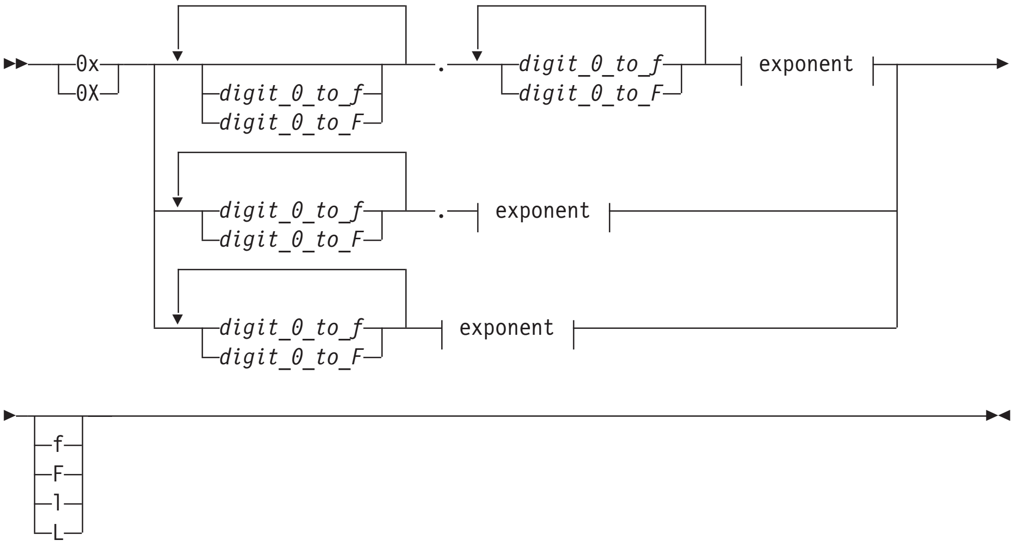
A hexadecimal floating constant consists of the following:
- the hexadecimal prefix
- a significant part
- a binary exponent part
- an optional suffix

C++ C++ extends Standard C++ and C++98 to support hexadecimal floating constants for compatibility with C99.

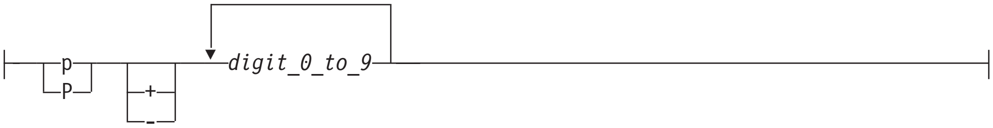
The significant part represents a rational number and is composed of the following:
- a sequence of hexadecimal digits (whole-number part)
- an optional fraction part

The optional fraction part is a period followed by a sequence of hexadecimal digits.

The exponent part indicates the power of 2 to which the significant part is raised, and is an optionally signed decimal integer. The type suffix is optional. The full syntax is as follows:



Exponent:



You can omit either the whole-number part or the fraction part, but not both. The binary exponent part is required to avoid the ambiguity of the type suffix F being mistaken for a hexadecimal digit.

Complex Literals

A complex literal type represents a complex number. The predefined macro `_Complex_I` represents a constant expression of type `const float _Complex` with the value of the imaginary unit. For example,

```
float _Complex varComplex = 2.0f + 2.0f*_Complex_I;
```

initializes the variable varComplex to type `float _Complex`.

The complex type can also be indicated by one of the suffixes: `i`, `I`, `j`, or `J`. The real part of the complex number can be indicated by one of the suffixes: `f`, `F`, `l`, or `L`. These suffixes are extensions of C99 for ease of porting applications developed with GNU C.

The simplified syntax for a complex literal is:

```
►►─┤ floating-constant ├─┤ complex-suffix ├──────────────────────────►◄
```

floating-constant:

```
├──┬─decimal-floating-constant─────┬──────────────────────────────────┤
   └─hexadecimal-floating-constant─┘
```

complex-suffix:

```
├──┬─┬─────────────────┬─suffixij─┬───────────────────────────────────┤
   │ └─floating-suffix─┘          │
   └─suffixij─┬─────────────────┬─┘
              └─floating-suffix─┘
```

where

floating-suffix is one of `f`, `F`, `l` (lowercase *el*) or `L`.

The suffixes `f` or `F` indicates a complex literal of type `float _Complex`. The suffixes `l` or `L` indicates a complex literal of type `long double _Complex`. A complex literal is of type `double _Complex` in the absense of suffixes.

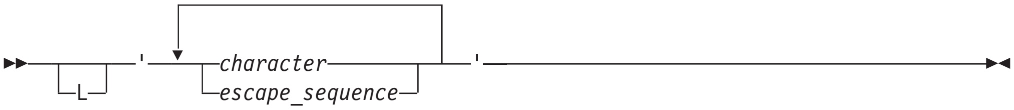
suffixij is one of `i`, `I`, `j`, `J`.

Related References
- "Unary operators for complex types" on page 120

Character Literals

A *character literal* contains a sequence of characters or escape sequences enclosed in single quotation mark symbols, for example `'c'`. A character literal may be prefixed with the letter `L`, for example `L'c'`. A character literal without the `L` prefix is an *ordinary character literal* or a *narrow character literal*. A character literal with the `L` prefix is a *wide character literal*. An ordinary character literal that contains more than one character or escape sequence (excluding single quotes (`'`), backslashes (\) or new-line characters) is a *multicharacter literal*.

Character literals have the following form:



At least one character or escape sequence must appear in the character literal. The characters can be from the source program character set, excluding the single quotation mark, backslash and new-line symbols. A character literal must appear on a single logical source line.

C A character literal has type **int**.

C++ A character literal that contains only one character has type **char**, which is an integral type.

In both C and C++, a wide character literal has type **wchar_t**, and a multicharacter literal has type **int**.

The value of a narrow or wide character literal containing a single character is the numeric representation of the character in the character set used at run time. The value of a narrow or wide character literal containing more than one character or escape sequence is implementation-defined.

You can represent the double quotation mark symbol by itself, but you must use the backslash symbol followed by a single quotation mark symbol (`\'` escape sequence) to represent the single quotation mark symbol.

You can represent the new-line character by the `\n` new-line escape sequence.

You can represent the backslash character by the \\ backslash escape sequence.

The following are examples of character literals:

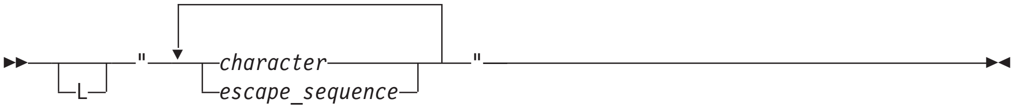
```
'a'
'\''
L'0'
'('
```

Related References

- “char and wchar_t Type Specifiers” on page 50
- “The Unicode Standard” on page 14

String Literals

A *string literal* contains a sequence of characters or escape sequences enclosed in double quotation mark symbols.



The universal character name for a character outside the basic source character set is allowed.

A string literal with the prefix **L** is a *wide string literal*. A string literal without the prefix **L** is an *ordinary* or *narrow string literal*.

▶ C The type of a narrow string literal is array of **char** and the type of a wide string literal is array of **wchar_t**.

▶ C++ The type of a narrow string literal is array of **const char** and the type of a wide string literal is array of **const wchar_t**. Both types have **static** storage duration.

The following are examples of string literals:

```
char titles[ ] = "Handel's \"Water Music\"";
char *mail_addr = "Last Name    First Name    MI   Street Address \
   City     Province   Postal code ";
char *temp_string = "abc" "def" "ghi";  /* *temp_string = "abcdefghi\0" */
wchar_t *wide_string = L"longstring";
```

A null ('\0') character is appended to each string. For a wide string literal, the value '\0' of type **wchar_t** is appended. By convention, programs recognize the end of a string by finding the null character.

Multiple spaces contained within a string literal are retained.

To continue a string on the next line, use the line continuation character (\ symbol) followed by optional whitespace and a new-line character (required). In the following example, the string literal second causes a compile-time error.

```
char *first = "This string continues onto the next\
  line, where it ends.";                /* compiles successfully.   */
char *second = "The comment makes the \ /* continuation symbol      */
  invisible to the compiler.";          /* compilation error.       */
```

Concatenation

Another way to continue a string is to have two or more consecutive strings. Adjacent string literals will be concatenated to produce a single string. If a wide string literal and a narrow string literal are adjacent to each other, the resulting behavior is undefined. The following example demonstrates this:

```
"hello " "there"     /* is equivalent to "hello there"                     */
"hello " L"there"    /* the behavior at the C89 language level is undefined */
"hello" "there"      /* is equivalent to "hellothere"                      */
```

Characters in concatenated strings remain distinct. For example, the strings ″\xab″ and ″3″ are concatenated to form ″\xab3″. However, the characters \xab and 3 remain distinct and are not merged to form the hexadecimal character \xab3.

▶ C If a wide string literal and a narrow string literal are adjacent, the result is a wide string literal.

Following any concatenation, '\0' of type **char** is appended at the end of each string. C++ programs find the end of a string by scanning for this value. For a wide string literal, '\0' of type **wchar_t** is appended. For example:

```
char *first = "Hello ";            /* stored as "Hello \0"       */
char *second = "there";            /* stored as "there\0"        */
char *third = "Hello " "there";    /* stored as "Hello there\0"  */
```

Compound Literals

A *compound literal* is a postfix expression that provides an unnamed object whose value is given by the initializer list. The expressions in the initializer list may be constants. The C99 language feature allows compound constants in initializers and expressions, providing a way to specify constants of aggregate or union type. When an instance of one of these types is used only once, a compound literal eliminates the necessity of temporary variables. C++ supports this feature as an extension to Standard C++ for compatibility with C.

The syntax for a compound literal resembles that of a cast expression. However, a compound literal is an lvalue, while the result of a cast expression is not. Furthermore, a cast can only convert to scalar types or **void**, whereas a compound literal results in an object of the specified type. The syntax is as follows:

```
►►─(─type_name─)─{──┬─initializer_list───┬─}──────────────────────────────►◄
                    └─initializer_list─,─┘
```

If the type is an array of unknown size, the size is determined by the initializer list.

A compound literal has **static** storage duration if it occurs outside the body of a function and the initializer list consists of constant expressions. Otherwise, it has automatic storage duration associated with the enclosing block. The following expressions have different meanings. The compound literals have automatic storage duration when they occur within the body of a function.:

```
"string"            /* an array of char with static storage duration */
(char[]){"string"}           /* modifiable     */
(const char[]){"string"}     /* not modifiable */
```

A **const**-qualified compound literal can be placed in read-only memory. Compound literals with **const**-qualified types can share storage with string literals with the same or overlapping representations. For example,

```
(const char[]){"string"} == "string"
```

might yield 1 if the storage is shared. However, compound literals with **const**-qualified types are not necessarily shared. The following expressions result in two distinct objects of type struct s.

```
(const struct s){1,2,3}
(const struct s){1,2,3}
```

Linux Mac OS X Compound literals having vector type follow the same semantic rules as regular compound literals:

```
vector unsigned int v1 = (vector unsigned int) {1,2,3,4};
```

A compound literal is not a constant expression and therefore cannot be used to initialize a static variable. However, for compatibility with GNU C/C++, a static variable having vector type can be initialized with a compound literal of the same vector type, provided that all the initializers in the initializer list are constant expressions.

Vector Literals

Linux Mac OS X A vector literal is a constant expression for which the value is interpreted as a vector type. The data type of a vector literal is represented by a parenthesized vector type, and its value is represented by a parenthesized set of constant expressions that represent the vector elements. When all vector elements have the same value, the value of the literal can be represented by a single parenthesized constant expression. Vector literals allow the initialization of vector types.

A vector literal is of the following form:

```
                           ┌─,─────────────────────┐
                           ▼                       │
►►──(──vector_type──)──(──────constant_expression──┴──)──────────────────►◄
```

where *vector_type* is a supported vector type.

The *constant_expression* or multiple constant expressions must be appropriate for the vector literal type. The type also determines how many comma-separated constant expressions must be present in the declaration.

When multiple constant expressions are specified, the number of constant expressions must be exactly:

4 For vector int, vector long, and vector float types.
8 For vector short and vector pixel types.
16 For vector char types.

A vector literal can be initialized through an initialization list. The syntax is:

```
►►──vector_type──identifier──=──{──┬─initializer_list───┬──}──;──────────►◄
                                   └─initializer_list ,─┘
```

Vector Literal Cast

A vector literal can be cast to another vector type. A vector literal cast does not change the bit pattern of the operand: the 128 bits representing the value remains the same before and after the cast.

Related References

- ″Initialization of vector types″ on page 79

Chapter 3. Declarations

A *declaration* establishes the names and characteristics of data objects and functions used in a program. A *definition* allocates storage for data objects or specifies the body for a function, and associates an identifier with that object or function. When you declare or define a type, no storage is allocated.

In diverse ways, declarations determine the interrelated attributes of an object: storage class, type, scope, visibility, storage duration, and linkage.

Declaration Overview

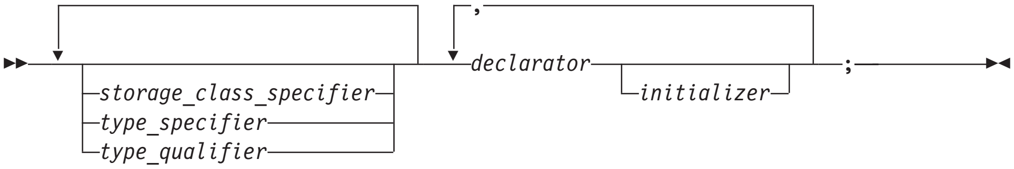
Declarations determine the following properties of data objects and their identifiers:

- Scope, which describes the region of program text in which an identifier can be used to access its object.
- Visibility, which describes the region of program text from which legal access can be made to the identifier's object.
- Duration, which defines the period during which the identifiers have real, physical objects allocated in memory.
- Linkage, which describes the correct association of an identifier to one particular object.
- Type, which determines how much memory is allocated to an object and how the bit patterns found in the storage allocation of that object should be interpreted by the program.

The lexical order of elements of a declaration for a data object is as follows:

- Storage duration and linkage specification
- Type specification
- Declarators, which introduce identifiers and make use of type qualifiers and storage qualifiers
- Initializers, which initialize storage with initial values

All data declarations have the form:



The following table shows examples of declarations and definitions. The identifiers declared in the first column do not allocate storage; they refer to a corresponding definition. In the case of a function, the corresponding definition is the code or body of the function. The identifiers declared in the second column allocate storage; they are both declarations and definitions.

Declarations	Declarations and Definitions
`extern double pi;`	`double pi = 3.14159265;`
`float square(float x);`	`float square(float x) { return x*x; }`

Declarations	Declarations and Definitions

`struct payroll;`	`struct payroll {`
	`char *name;`
	`float salary;`
	`} employee;`

Related References
- Chapter 4, "Declarators," on page 81

Variable Attributes

Linux Variable attributes are orthogonal language extensions provided to facilitate handling programs developed with the GNU C/C++ compilers. These language features allow you use named attributes to modify the declarations of variables. The syntax and supported variable attributes are described in this section. For unsupported attribute names, the IBM C/C++ compilers issue diagnostics and ignore the attribute specification.

The keyword **__attribute__** specifies a variable attribute. An attribute syntax has the general form:

```
                    ┌─,─────────────────┐
                    ▼                   │
►►──__attribute__──((───┬─attribute_name─────┬─┴──))──────────────►◄
                        └─__attribute_name__─┘
```

Attribute specifiers are declaration specifiers, and therefore can appear before the declarator in a declaration. The attribute specifier can also follow a declarator. In this case, it applies only to that particular declarator in a comma-separated list of declarators.

A variable attribute specification using the form *__attribute_name__* (that is, the variable attribute keyword with double underscore characters leading and trailing) reduces the likelihood of a name conflict with a macro of the same name.

Related References
- "Function Attributes" on page 163
- "Type Attributes" on page 45

The aligned Variable Attribute

Linux The variable attribute **aligned** allows you to specify a minimum alignment in bytes for a variable or structure member. Specifying the alignment can improve the efficiency of copy operations because the compiler can then use the instructions that copy the largest amounts of memory when copying to or from the variables or structure members aligned in this way.

When the **aligned** variable attribute is applied to an automatic variable, the alignment is limited by the maximum alignment of the stack. When attribute **aligned** is applied to a bit field structure member, the bit field container is aligned according to the alignment specification, unless the alignment of the container is greater than the alignment factor. In this case, attribute **aligned** is ignored.

```
►►──__attribute__──((──┬─aligned─────┬──┬──────────────────────┬──))──────►◄
                       └─__aligned__─┘  └─(─alignment_factor─)─┘
```

where *alignment_factor* is a constant expression that evaluates to a positive power of 2.

Omitting the alignment factor (and its enclosing parentheses) allows the compiler to determine an alignment. The alignment will be the largest strict alignment for any natural type (that is, integral or real) that can be handled on the target machine.

The **aligned** attribute only increases alignment. The **packed** attribute can be used to decrease it. An alignment factor greater than the platform maximum is ignored with a warning, and the results are unpredictable.

The init_priority Variable Attribute

Linux C++ The variable attribute **init_priority** is an orthogonal extension to C++ that allows you to control the initialization order of objects defined in namespace scope within a single compilation unit. The attribute takes a parameter indicating the relative priority of initialization. A lower number indicates a higher priority.

The syntax is as follows:

```
►►──type_specifier──declarator──__attribute__──────────────────────────────►

►──((──┬─init_priority─────┬──┬──────────────────────────┬──))──────────────►◄
       └─__init_priority__─┘  └─(──relative_priority──)─┘
```

where *relative_priority* is a constant integral expression between 101 and 65535, inclusive.

The mode Variable Attribute

Linux The variable attribute **mode** allows you to override the type specifier in a variable declaration. The original type indicated by the type specifier is overridden by an integral type of a particular size. The size is indicated by the value of the mode parameter. For example, a mode value of __word__ results in an integer variable that is four bytes in size. The sign of the original type specifier is preserved.

Valid arguments for attribute **mode** are `byte`, `word`, and `pointer`, and the forms of these modes with leading and trailing double underscores.

- `byte` means a 1-byte integer type
- `word` means a 4-byte integer type
- `pointer` means 4-byte integer type in 32-bit mode and an 8-byte integer type in 64-bit mode

The syntax is as follows:

```
►►──__attribute__──((──┬─mode─────┬──(──┬─byte────────┬──)──))──────────►◄
                       └─__mode__─┘     ├─word────────┤
                                        ├─pointer─────┤
                                        ├─__byte__────┤
                                        ├─__word__────┤
                                        └─__pointer__─┘
```

where *mode* is a type specifier that includes an indication of width.

The packed Variable Attribute

Linux The variable attribute **packed** allows you to specify that a structure member or bit field structure member should have the smallest possible alignment: one byte for a member and one bit for a bit field member, unless a larger value is specified with the **aligned** variable attribute.

The syntax is as follows:

```
►►──__attribute__──((──┬─packed─────┬─))────────────────────────────────►◄
                       └─__packed__─┘
```

The weak Variable Attribute

Linux The variable attribute **weak** and the function attribute **weak** have the same behavior and rationale. The syntax for applying an attribute specifier to a variable declaration allows the variable attribute specifier to appear either before or after the declarator. The following diagrams show the two forms of valid declaration syntax.

```
►►──type_specifier──__attribute__──((──┬─weak─────┬─))──variable_name──────►◄
                                       └─__weak__─┘
```

The above syntax is the same as that for declaring and defining a function **weak**. The other valid syntax for declaring a **weak** variable is the same as that for a **weak** function declaration, but not the function definition.

```
►►──type_specifier──variable_name──__attribute__──((──┬─weak─────┬─))──────►◄
                                                      └─__weak__─┘
```

The __align Specifier

Linux The **__align** keyword allows you to specify an explicit alignment for a data structure. The keyword is an orthogonal language extension intended to be used in the definition of an aggregate type or in the declaration of a first-level variable. The specified byte boundary affects the alignment of an aggregate as a whole, not that of its members. The **__align** specifier can be applied to an aggregate definition nested within another aggregate definition, but not to individual elements of an aggregate or class. The alignment specification is ignored for parameters and automatic variables.

A declaration takes one of the following forms:

```
►►──declarator──__align──(──int_constant──)──identifier──;──────────────►◄
```

Structure or union syntax:

```
►►──__align──(──int_constant──)──struct_or_union_specifier──┬─────┬──{──struct_declaration_list──}──;──────►◄
                                                            └─tag─┘
```

where:

int_constant
: Is a positive integer value indicating the byte-alignment boundary. The legal values are 1, 2, 4, 8, or 16.

struct_or_union_specifier
: Is a structure or union specifier.

struct_declaration_list
Is a structure declaration list.

tag Is a structure or union identifier.

Restrictions and limitations

The **__align** specifier cannot be used where the size of the variable alignment is smaller than the size of the type alignment.

Not all alignments may be representable in an object file.

The **__align** specifier cannot be applied to the following:

- Individual elements within an aggregate definition.
- Individual elements of an array.
- Variables of incomplete type.
- Aggregates declared but not defined.
- Other types of declarations or definitions, such as a typedef, a function, or an enumeration.

Tentative Definitions

C A *tentative definition* is any external data declaration that has no storage class specifier and no initializer. A tentative definition becomes a full definition if the end of the translation unit is reached and no definition has appeared with an initializer for the identifier. In this situation, the compiler reserves uninitialized space for the object defined.

The following statements show normal definitions and tentative definitions.

```
int i1 = 10;         /* definition, external linkage */
static int i2 = 20;  /* definition, internal linkage */
extern int i3 = 30;  /* definition, external linkage */
int i4;              /* tentative definition, external linkage */
static int i5;       /* tentative definition, internal linkage */

int i1;              /* valid tentative definition */
int i2;              /* not legal, linkage disagreement with previous */
int i3;              /* valid tentative definition */
int i4;              /* valid tentative definition */
int i5;              /* not legal, linkage disagreement with previous */
```

C++ C++ does not support the concept of a tentative definition: an external data declaration without a storage class specifier is always a definition.

Related References

- “Declaration Overview” on page 31
- “Storage Class Specifiers” on page 36

Objects

An *object* is a region of storage that contains a value or group of values. Each value can be accessed using its identifier or a more complex expression that refers to the object. In addition, each object has a unique *data type*. Both the identifier and data type of an object are established in the object *declaration*.

The data type of an object determines the initial storage allocation for that object and the interpretation of the values during subsequent access. It is also used in any type checking operations.

Both C and C++ have built-in, or *fundamental*, data types and user-defined data types. Standard data types include signed and unsigned integers, floating-point numbers, and characters. User-defined types include enumerations, structures, and unions. All C++ classes are considered user-defined types.

An instance of a class type is commonly called a *class object*. The individual class members are also called objects. The set of all member objects comprises a class object.

Related References
- "Lvalues and Rvalues" on page 103
- Chapter 12, "Classes," on page 253

Storage Class Specifiers

A storage class specifier is used to refine the declaration of a variable, a function, and parameters. The storage class specifier used within the declaration determines whether:
- The object has internal, external, or no linkage
- The object is to be stored in memory or in a register, if available
- The object receives the default initial value `0` or an indeterminate default initial value
- The object can be referenced throughout a program or only within the function, block, or source file where the variable is defined
- The storage duration for the object is *static* (storage is maintained throughout program run time) or *automatic* (storage is maintained only during the execution of the block where the object is defined)

For a variable, its default storage duration, scope, and linkage depend on where it is declared: whether inside or outside a block statement or the body of a function. When these defaults are not satisfactory, you can specify an explicit storage class: **auto**, **static**, **extern**, or **register**. In C++, you have the additional option of being able to specify the storage class **mutable** for a class data member to make it modifiable, even though the member is part of an object that has been declared **const**.

For a function, the storage class specifier determines the linkage of the function. The only options are **extern** and **static**. A function that is declared with the **extern** storage class specifier has external linkage, which means that it can be called from other translation units. A function declared with the **static** storage class specifier has internal linkage, which means that it may be called only within the translation unit in which it is defined. The default for a function is external linkage.

▶ C The only storage class that can be specified for a function parameter is **register**. The reason is that function parameters have the same properties as `auto` variables: automatic storage duration, block scope, and no linkage.

Declarations with the **auto** or **register** storage class specifier result in automatic storage. Those with the **static** storage class specifier result in static storage.

Most local declarations that do not include the **extern** storage class specifier allocate storage; however, function declarations and type declarations do not allocate storage.

The only storage class specifiers allowed in a namespace or global scope declaration are **static** and **extern**. In C++, the use of **static** for a specifying internal linkage is deprecated. Use the unnamed namespace instead.

The storage class specifiers in C and C++ are:

- **auto**
- **extern**
- ▶ C++ **mutable**
- **register**
- **static**
- **typedef**

typedef is categorized as a storage class specifier because of similarities in syntax rather than functionality and because a **typedef** declaration does not allocate storage.

auto Storage Class Specifier

The **auto** storage class specifier lets you explicitly declare a variable with *automatic storage*. The `auto` storage class is the default for variables declared inside a block. A variable `x` that has automatic storage is deleted when the block in which `x` was declared exits.

You can only apply the **auto** storage class specifier to names of variables declared in a block or to names of function parameters. However, these names by default have automatic storage. Therefore the storage class specifier **auto** is usually redundant in a data declaration.

Initialization

You can initialize any **auto** variable except parameters. If you do not explicitly initialize an automatic object, its value is indeterminate. If you provide an initial value, the expression representing the initial value can be any valid C or C++ expression. The object is then set to that initial value each time the program block that contains the object's definition is entered.

Note that if you use the **goto** statement to jump into the middle of a block, automatic variables within that block are not initialized.

Storage duration

Objects with the **auto** storage class specifier have automatic storage duration. Each time a block is entered, storage for **auto** objects defined in that block is made available. When the block is exited, the objects are no longer available for use. An object declared with no linkage specification and without the **static** storage class specifier has automatic storage duration.

If an **auto** object is defined within a function that is recursively invoked, memory is allocated for the object at each invocation of the block.

Linkage

An `auto` variable has block scope and no linkage.

Related References
- “Block Statement” on page 192
- “goto Statement” on page 206
- “Function Declarations” on page 160

extern Storage Class Specifier

The **extern** storage class specifier lets you declare objects and functions that several source files can use. An **extern** variable, function definition, or declaration makes the described variable or function usable by the succeeding part of the current source file. This declaration does not replace the definition. The declaration is used to describe the variable that is externally defined.

An **extern** declaration can appear outside a function or at the beginning of a block. If the declaration describes a function or appears outside a function and describes an object with external linkage, the keyword **extern** is optional. If you do not specify a storage class specifier, the function is assumed to have external linkage.

If a declaration for an identifier already exists at file scope, any **extern** declaration of the same identifier found within a block refers to that same object. If no other declaration for the identifier exists at file scope, the identifier has external linkage.

It is an error to include a declaration for the same function with the storage class specifier **static** before the declaration with no storage class specifier because of the incompatible declarations. Including the **extern** storage class specifier on the original declaration is valid and the function has internal linkage.

▶ Linux When the GNU C semantics for inline functions are desired and source code is compiled accordingly, the keyword **extern** combines with the keyword **inline** to behave as a single keyword.

Related References
- “The extern inline keyword” on page 186

▶ C++ The following remarks pertain to C++ only:
- C++ restricts the use of the **extern** storage class specifier to the names of objects or functions. Using the **extern** specifier with type declarations is illegal.
- In C++, an **extern** declaration cannot appear in class scope.

Initialization

You can initialize any object with the **extern** storage class specifier at global scope in C or at namespace scope in C++. The initializer for an **extern** object must either:
- Appear as part of the definition and the initial value must be described by a constant expression. OR
- Reduce to the address of a previously declared object with static storage duration. You may modify this object with pointer arithmetic. (In other words, you may modify the object by adding or subtracting an integral constant expression.)

If you do not explicitly initialize an **extern** variable, its initial value is zero of the appropriate type. Initialization of an **extern** object is completed by the time the program starts running.

Storage duration

All **extern** objects have static storage duration. Memory is allocated for **extern** objects before the `main` function begins running, and is freed when the program terminates. The scope of the variable depends on the location of the declaration in the program text. If the declaration appears within a block, the variable has block scope; otherwise, it has file scope.

Linkage

▶ C Like the scope, the linkage of a variable declared **extern** depends on the placement of the declaration in the program text. If the variable declaration appears outside of any function definition and has been declared **static** earlier in the file, the variable has internal linkage; otherwise, it has external linkage in most cases. All object declarations that occur outside a function and that do not contain a storage class specifier declare identifiers with external linkage. All function definitions that do not specify a storage class define functions with external linkage.

▶ C++ For objects in the unnamed namespace, the linkage may be external, but the name is unique, and so from the perspective of other translation units, the name effectively has internal linkage.

Related References
- “External Linkage” on page 7
- “Internal Linkage” on page 6
- “static Storage Class Specifier” on page 40
- “Class Scope” on page 4
- Chapter 10, “Namespaces,” on page 229
- “Inline Functions” on page 186

mutable Storage Class Specifier

▶ C++ The **mutable** storage class specifier is used only on a class data member to make it modifiable even though the member is part of an object declared as **const**. You cannot use the mutable specifier with names declared as **static** or **const**, or reference members.

```
class A
{
  public:
    A() : x(4), y(5) { };
    mutable int x;
    int y;
};

int main()
{
  const A var2;
  var2.x = 345;
  // var2.y = 2345;
}
```

In this example, the compiler would not allow the assignment `var2.y = 2345` because `var2` has been declared as `const`. The compiler will allow the assignment `var2.x = 345` because `A::x` has been declared as **mutable**.

register Storage Class Specifier

The **register** storage class specifier indicates to the compiler that the value of the object should reside in a machine register. The compiler is not required to honor this request. Because of the limited size and number of registers available on most systems, few variables can actually be put in registers. If the compiler does not allocate a machine register for a **register** object, the object is treated as having the storage class specifier **auto**. A **register** storage class specifier indicates that the object, such as a loop control variable, is heavily used and that the programmer hopes to enhance performance by minimizing access time.

An object having the **register** storage class specifier must be defined within a block or declared as a parameter to a function.

Initialization

You can initialize any **register** object except parameters. If you do not initialize an automatic object, its value is indeterminate. If you provide an initial value, the expression representing the initial value can be any valid C or C++ expression. The object is then set to that initial value each time the program block that contains the object's definition is entered.

Storage duration

Objects with the **register** storage class specifier have automatic storage duration. Each time a block is entered, storage for **register** objects defined in that block is made available. When the block is exited, the objects are no longer available for use.

If a **register** object is defined within a function that is recursively invoked, memory is allocated for the variable at each invocation of the block.

Linkage

Since a **register** object is treated as the equivalent to an object of the **auto** storage class, it has no linkage.

▶ C **Restrictions**

- The **register** storage class specifier is legal only for variables declared in a block. You cannot use it in global scope data declarations.
- A register does not have an address. Therefore, you cannot apply the address operator (&) to a **register** variable.

▶ C++ **Restrictions**

- You cannot use the **register** storage class specifier when declaring objects in namespace scope.
- Unlike C, C++ lets you take the address of an object with the **register** storage class. For example:

```
register int i;
int* b = &i;      // valid in C++, but not in C
```

static Storage Class Specifier

The **static** storage class specifier lets you define objects or functions with internal linkage, which means that each instance of a particular identifier represents the same object or function within one file only. In addition, objects declared **static** have *static storage duration*, which means that memory for these objects is allocated when the program begins running and is freed when the program terminates.

Static storage duration for an object is different from file or global scope: an object can have static duration but local scope. On the other hand, the **static** storage class specifier can be used in a function declaration only if it is at file scope.

The **static** storage class specifier can only be applied to the following names:
- Objects
- Functions
- Class members
- Anonymous unions

You cannot declare any of the following as **static**:
- Type declarations
- Function declarations within a block
- Function parameters

C The keyword **static** is the major mechanism in C to enforce information hiding. C++ enforces information hiding through the namespace language feature and the access control of classes.

At the C99 language level, the **static** keyword can be used in the declaration of an array parameter to a function. The **static** keyword indicates that the argument passed into the function is a pointer to an array of at least the specified size. In this way, the compiler is informed that the pointer argument is never null.

C++ The use of the keyword **static** to limit the scope of external variables is deprecated for declaring objects in namespace scope.

Initialization

You initialize a **static** object with a constant expression, or an expression that reduces to the address of a previously declared **extern** or **static** object, possibly modified by a constant expression. If you do not explicitly initialize a **static** (or external) variable, it will have a value of zero of the appropriate type.

C More precisely, in C,
- If the variable is a pointer type, it is initialized to a null pointer.
- If it has arithmetic type, it is initialized to positive or unsigned zero.
- If it is an aggregate, the first named member is recursively initialized according to these rules.
- If it is a union, the first named member is recursively initialized according to these rules.

A **static** variable in a block is initialized only one time, prior to program execution, whereas an **auto** variable that has an initializer is initialized every time it comes into existence.

Each time a recursive function is called, it gets a new set of **auto** variables. However, if the function has a **static** variable, the same storage location is used by all calls of the function.

C++ A static object of class type will use the default constructor if you do not initialize it. Automatic and register variables that are not initialized will have undefined values.

In C++, you may initialize a **static** object with a non-constant expression, but the following usage has been deprecated:

```
static int staticInt = 5;
int main()
{
   // . . .
}
```

C++ provides the namespaces language feature to limit the scope of external variables.

Linkage

A declaration of an object or file that contains the **static** storage class specifier and has file scope, gives the identifier internal linkage. Each instance of the particular identifier therefore represents the same object or function within one file only.

Example

Suppose a static variable x has been declared in function f(). When the program exits the scope of f(), x is not destroyed. The following example demonstrates this:

```
#include <stdio.h>

int f(void) {
  static int x = 0;
  x++;
  return x;
}

int main(void) {
  int j;
  for (j = 0; j < 5; j++) {
    printf("Value of f(): %d\n", f());
  }
  return 0;
}
```

The following is the output of the above example:

```
Value of f(): 1
Value of f(): 2
Value of f(): 3
Value of f(): 4
Value of f(): 5
```

Because x is a static variable, it is not reinitialized to 0 on successive calls to f().

typedef

A **typedef** declaration lets you define your own identifiers that can be used in place of type specifiers such as **int**, **float**, and **double**. A **typedef** declaration does not reserve storage. The names you define using **typedef** are not new data types, but synonyms for the data types or combinations of data types they represent. The name space for a **typedef** name is the same as other identifiers. The exception to this rule is if the **typedef** name specifies a variably modified type. In this case, it has block scope.

When an object is defined using a **typedef** identifier, the properties of the defined object are exactly the same as if the object were defined by explicitly listing the data type associated with the identifier.

Linux The **typedef** storage-class specifier has been extended to handle vector types, provided that AltiVec language extensions have been enabled. A vector type can be used in a **typedef** declaration, and the new type name can be used in the usual ways, except for declaring other vectors. In a vector declaration context, a **typedef** name is disallowed as a type specifier. The following example illustrates a typical usage of **typedef** with vector types:

```
typedef vector unsigned short vint16;
vint16 v1;
```

Examples of typedef Declarations

The following statements declare `LENGTH` as a synonym for **int** and then use this **typedef** to declare `length`, `width`, and `height` as integer variables:

```
typedef int LENGTH;
LENGTH length, width, height;
```

The following declarations are equivalent to the above declaration:

```
int length, width, height;
```

Similarly, **typedef** can be used to define a class type (structure, union, or C++ class). For example:

```
typedef struct {
                int scruples;
                int drams;
                int grains;
               } WEIGHT;
```

The structure `WEIGHT` can then be used in the following declarations:

```
WEIGHT  chicken, cow, horse, whale;
```

In the following example, the type of `yds` is "pointer to function with no parameter specified, returning int".

```
typedef int SCROLL();
extern SCROLL *yds;
```

In the following typedefs, the token `struct` is part of the type name: the type of `ex1` is `struct a`; the type of `ex2` is `struct b`.

```
typedef struct a { char x; } ex1, *ptr1;
typedef struct b { char x; } ex2, *ptr2;
```

Type `ex1` is compatible with the type `struct a` and the type of the object pointed to by `ptr1`. Type `ex1` is not compatible with `char`, `ex2`, or `struct b`.

C++ The remainder of this section pertains to C++ only.

In C++, a **typedef** name must be different from any class type name declared within the same scope. If the **typedef** name is the same as a class type name, it can only be so if that **typedef** is a synonym of the class name. This condition is not the same as in C. The following can be found in standard C headers:

```
typedef class C { /*  data and behavior  */ } C;
```

A C++ class defined in a **typedef** without being named is given a dummy name and the **typedef** name for linkage. Such a class cannot have constructors or destructors. For example:

```
typedef class {
               Trees();
              } Trees;
```

Here the function `Trees()` is an ordinary member function of a class whose type name is unspecified. In the above example, `Trees` is an alias for the unnamed class, not the class type name itself, so `Trees()` cannot be a constructor for that class.

Type Specifiers

Type specifiers indicate the type of the object or function being declared. The following are the available kinds of type specifiers:

- Simple type specifiers
- Enumerated specifiers
- **const** and **volatile** qualifiers
- C++ Class specifiers
- C++ Elaborated type specifiers

The term *scalar types* collectively refers in C to arithmetic types or pointer types. In C++, scalar types include all the cv-qualified versions of the C scalar types, plus all the cv-qualified versions of enumeration and pointer-to-member types.

The term *aggregate type* refers in both C and C++ to array and structure types.

C++ In C++, types must be declared in declarations. They may not be declared in expressions.

Type Names

A data type, more precisely, a *type name*, is required in several contexts as something that you must specify without declaring an object; for example, when writing an explicit cast expression or when applying the `sizeof` operator to a type. Syntactically, the name of a data type is the same as a declaration of a function or object of that type, but without the identifier.

To read or write a type name correctly, put an ″imaginary″ identifier within the syntax, splitting the type name into simpler components. For example, **int** is a type specifier, and it always appears to the left of the identifier in a declaration. An imaginary identifier is unnecessary in this simple case. However, `int *[5]` (an array of 5 pointers to **int**) is also the name of a type. The type specifier **int *** always appears to the left of the identifier, and the array subscripting operator always appears to the right. In this case, an imaginary identifier is helpful in distinguishing the type specifier.

As a general rule, the identifier in a declaration always appears to the left of the subscripting and function call operators, and to the right of a type specifier, type qualifier, or indirection operator. Only the subscripting, function call, and indirection operators may appear in a declaration. They bind according to normal operator precedence, which is that the indirection operator is of lower precedence than either the subscripting or function call operators, which have equal ranking in the order of precedence. Parentheses may be used to control the binding of the indirection operator.

It is possible to have a type name within a type name. For example, in a function type, the parameter type syntax nests within the function type name. The same rules of thumb still apply, recursively.

The following constructions illustrate applications of the type naming rules.

```
int *[5]       /* array of 5 pointers to int                           */
int (*)[5]     /* pointer to an array of 5 ints                        */
int (*)[*]     /* pointer to an variable length array of
                    an unspecified number of ints                      */
int *()        /* function with no parameter specification
                    returning a pointer to int                         */
int (*)(void)  /* function with no parameters returning an int         */
int (*const [])(unsigned int, ...)
               /* array of an unspecified number of
                    constant pointers to functions returning an int
                    Each function takes one parameter of type unsigned int
                    and an unspecified number of other parameters       */
```

The compiler turns any function designator into a pointer to the function. This behavior simplifies the syntax of function calls.

```
int foo(float);   /* foo is a function designator */
int (*p)(float);  /* p is a pointer to a function */
p=&foo;           /* legal, but redundant         */
p=foo;            /* legal because the compiler turns foo into a function pointer */
```

C++ In C++, the keywords **typename** and **class**, which are interchangeable, indicate the name of the type.

Type Attributes

Linux A type attribute is a declaration specifier that uses the keyword **__attribute__** and its accompanying syntax to specify special properties for a structure, union, enumeration, or class. Type attributes are orthogonal extensions to C and C++, implemented to facilitate porting programs developed with GNU C and C++.

The syntax of a type attribute is of the general form:

```
►►──__attribute__──((──┬─attribute_name─────┬─))──────────────────────────►◄
                       └─__attribute_name__─┘
```

Type Attribute aligned

Linux Type attribute `aligned` allows you to specify the alignment of a structure, class, union, or enumeration. The syntax and considerations for specifying alignment factor are the same as for variable attribute `aligned`. Like variable attribute `aligned`, type attribute `aligned` can only increase alignment. Type attribute `packed` is used to decrease alignment.

If the attribute appears immediately after the class, struct, union, or enumeration token or immediately after the closing right curly brace, it applies to the type identifier. It can also be specified on a **typedef** declaration. In a variable declaration, such as

```
class A {} a;
```

the placement of the type attribute can be confusing.

In the following definitions, the attribute applies to `A`:

```
struct __attribute__((__aligned__(8))) A {};
struct A {} __attribute__((__aligned__(8))) ;
struct __attribute__((__aligned__(8))) A {} a;
struct A {} __attribute__((__aligned__(8))) a;
```

```
typedef struct __attribute__((__aligned__(8))) A {} a;
typedef struct A {} __attribute__((__aligned__(8))) a;
```

In the following definitions, the attribute applies to a:

```
__attribute__((__aligned__(8))) struct A {} a;
struct A {} const __attribute__((__aligned__(8))) a;

__attribute__((__aligned__(8))) typedef struct A {} a;
typedef __attribute__((__aligned__(8))) struct A {} a;
typedef struct A {} const __attribute__((__aligned__(8))) a;
typedef struct A {} a __attribute__((__aligned__(8)));
```

Type Attribute packed

Linux Specifying the packed type attribute on a struct, class, union, or enumeration type indicates that the minimum amount of required memory is to be used for that type. Placement of type attribute packed is the same as for type attribute aligned, except that type attribute packed is not allowed on a **typedef** declaration.

Type Attribute transparent_union

Linux The transparent_union attribute applied to a union definition or a union **typedef** indicates the union can be used as a *transparent union*. Whenever a transparent union is the type of a function parameter and that function is called, the transparent union can accept an argument of any type that matches that of one of its members without an explicit cast. Arguments to this function parameter are passed to the transparent union, using the calling convention of the first member of the union type. Because of this, all members of the union must have the same machine representation. Transparent unions are useful in library functions that use multiple interfaces to resolve issues of compatibility. The language feature is an orthogonal extension to C89, C99, and Standard C++, and has been implemented to facilitate porting programs originally developed with GNU C.

The type attribute must follow the closing brace of the union or **typedef** definition.

```
union u_t {
   int a;
   short b;
   char c;
} __attribute__((__transparent_union__)) U;

typedef union {
   int *iptr;
   union u2_t *u2ptr;
} status_ptr_t  __attribute__((__transparent_union__));
```

Type attribute transparent_union can be applied to anonymous unions with tag names.

When type attribute transparent_union is applied to the outer union of a nested union, the size of the inner union (that is, its largest member) is used to determine if it has the same machine representation as the other members of the outer union. For example,

```
union u_t {
   union u2_t {
      char a;
      short b;
      char c;
      char d;
   };
   int a;
} __attribute__((__transparent_union__));
```

attribute `transparent_union` is ignored because the first member of union `u_t`, which is itself a union, has a machine representation of 2 bytes, whereas the other member of union `u_t` is of type **int**, which has a machine representation of 4 bytes.

The same rationale applies to members of a union that are structures. When a member of a union to which type attribute `transparent_union` has been applied is a **struct**, the machine representation of the entire **struct** is considered, rather than members.

Restrictions

The union must be a complete union type

All members of the union must have the same machine representation as the first member of the union. This means that all members must be representable by the same amount memory as the first member of the union. The machine representation of the first member represents the maximum memory size for any remaining union members. For instance, if the first member of a union to which type attribute `transparent_union` has been applied is of type **int**, then all following members must be representable by at most 4 bytes. Members that are representable by 1, 2, or 4 bytes are considered valid for this transparent union.

The first member of the union cannot be a floating-point type (**float**, **double**, **float _Complex**, or **double _Complex**) or a **Vector** type. However, **float _Complex**, **double _Complex**, and **Vector** types can be members of a transparent union, as long as they are not the first member. The restriction that all members of the transparent union have the same machine representation as the first member still applies.

Examples

This example shows how attribute `transparent_union` can be applied to a function parameter declaration:

```
void foo( union u_t { int a; short b; char c;} __attribute__((transparent_union)) uu)
{
   printf("uu.b is %d\n",uu.b);
}

int main() {
   short s = 99;
   foo(s);
   return 0;
}
```

This example shows how Complex types can be members of a transparent union:

```
union u_t {
   int i[2];      // This member must has a machine representation of 8 bytes.
   float _Complex cf;
} __attribute__((__transparent_union__)) U;

void foo(union u_t uu) {
   printf("uu.cf is %f\n",uu.cf);
}
```

```
int main() {
   float _Complex my_cf = 5.0f + 1.0f * __I;
   foo(my_cf);
   return 0;
}
```

Compatible Types

▶ C The concept of compatible types combines the notions of being able to use two types together without modification (as in an assignment expression), being able to substitute one for the other without modification, and uniting them into a composite type. A *composite type* is that which results from combining two compatible types. Determining the resultant composite type for two compatible types is similar to following the usual binary conversions of integral types when they are combined with some arithmetic operators.

Obviously, two types that are the same are compatible; their composite type is the same type. Less obvious are the rules governing type compatibility of non-identical types, function prototypes, and type-qualified types. Names in `typedef` definitions are only synonyms for types, and so `typedef` names can possibly indicate identical and therefore compatible types. Pointers, functions, and arrays with certain properties can also be compatible types.

Identical Types

The presence of type specifiers in various combinations for arithmetic types may or may not indicate different types. For example, the type **signed int** is the same as **int**, except when used as the types of bit fields; but **char**, **signed char**, and **unsigned char** are different types.

The presence of a type qualifier changes the type. That is, **const int** is not the same type as **int**, and therefore the two types are not compatible.

Two arithmetic types are compatible only if they are the same type.

Compatibility Across Separately Compiled Source Files

The definition of a structure, union, or enumeration results in a new type. When the definitions for two structures, unions, or enumerations are defined in separate source files, each file can theoretically contain a different definition for an object of that type with the same name. The two declarations must be compatible, or the run time behavior of the program is undefined. Therefore, the compatibility rules are more restrictive and specific than those for compatibility within the same source file. For structure, union, and enumeration types defined in separately compiled files, the composite type is the type in the current source file.

The requirements for compatibility between two structure, union, or enumerated types declared in separate source files are as follows:

- If one is declared with a tag, the other must also be declared with the same tag.
- If both are completed types, their members must correspond exactly in number, be declared with compatible types, and have matching names.

For enumerations, corresponding members must also have the same values.

For structures and unions, the following additional requirements must be met for type compatibility:

- Corresponding members must be declared in the same order (applies to structures only).
- Corresponding bit fields must have the same widths.

C++ A separate notion of type compatibility as distinct from being of the same type does not exist in C++. Generally speaking, type checking in C++ is stricter than in C: identical types are required in situations where C would only require compatible types.

Simple Type Specifiers

A *simple type specifier* either specifies a (previously declared) user-defined type or a *fundamental type*. A fundamental type is a one that is built into the language. The following outline shows the categories of fundamental types:

- Arithmetic types
  - Integral types
    - C++ **bool**
    - **char**
    - **wchar_t**
    - Signed integer types
      - **signed char**
      - **short int**
      - **int**
      - **long int**
      - **long long int**
    - Unsigned integer types
      - C **_Bool**
      - **unsigned char**
      - **unsigned short int**
      - **unsigned int**
      - **unsigned long int**
      - **unsigned long long int**
  - Floating-point types
    - **float**
    - **double**
    - **long double**
- **void**

The floating point types are referred to as *real floating types* when there is a need to distinguish them from the complex types `float _Complex`, `double _Complex`, and `long double _Complex`. Collectively, the real floating and complex types are called the *floating types*.

Boolean Variables

A Boolean variable can be used to hold the integer values 0 or 1, or the C++ literals `true` and `false`, which are implicitly promoted to the integers 0 and 1 whenever an arithmetic value is necessary. The type specifier to declare a Boolean variable is **bool** in C++. To declare a Boolean variable in C, use the `bool` macro, which is defined in the header file `<stdbool.h>`. A Boolean variable may not be further qualified by the specifiers `signed`, `unsigned`, `short`, or `long`.

C The Boolean type is unsigned and has the lowest ranking in its category of standard unsigned integer types. In simple assignments, if the left operand is a

Boolean type, then the right operand must be either an arithmetic type or a pointer. An object declared as a Boolean type uses 1 byte of storage space, which is large enough to hold the values 0 or 1.

In C, a Boolean type can be used as a bit field type. If a nonzero-width bit field of Boolean type holds the value 0 or 1, then the value of the bit-field compares equal to 0 or 1, respectively.

Linux C The token **bool** is recognized as a keyword only when used in a vector declaration context and when the AltiVec language extensions have been enabled.

Linux _Bool uses 4 bytes of storage space, and is 4-byte aligned.

C++ Variables of type **bool** can hold either one of two values: `true` or `false`. An rvalue of type **bool** can be promoted to an integral type. A **bool** rvalue of `false` is promoted to the value 0, and a **bool** rvalue of `true` is promoted to the value 1.

The result of the equality, relational, and logical operators is of type **bool**: either of the Boolean constants `true` or `false`.

Use the type specifier **bool** and the literals `true` and `false` to make boolean logic tests. A *boolean logic test* is used to express the results of a logical operation. For example:

```
bool f(int a, int b)
{
  return a==b;
}
```

If `a` and `b` have the same value, `f()` returns true. If not, `f()` returns false.

char and wchar_t Type Specifiers

The **char** specifier has the following syntax:

```
►►──┬──────────┬──char──────────────────────────────────────────►◄
    ├─unsigned─┤
    └─signed───┘
```

The **char** specifier is an integral type.

A **char** has enough storage to represent a character from the basic character set. The amount of storage allocated for a **char** is implementation-dependent.

You initialize a variable of type **char** with a character literal (consisting of one character) or with an expression that evaluates to an integer.

Use **signed char** or **unsigned char** to declare numeric variables that occupy a single byte.

C++ For the purposes of distinguishing overloaded functions, a C++ **char** is a distinct type from **signed char** and **unsigned char**.

Examples of the char Type Specifier

The following example defines the identifier `end_of_string` as a constant object of type **char** having the initial value `\0` (the null character):

```
const char end_of_string = '\0';
```

The following example defines the **unsigned char** variable `switches` as having the initial value 3:

```
unsigned char switches = 3;
```

The following example defines `string_pointer` as a pointer to a character:

```
char *string_pointer;
```

The following example defines `name` as a pointer to a character. After initialization, `name` points to the first letter in the character string `"Johnny"`:

```
char *name = "Johnny";
```

The following example defines a one-dimensional array of pointers to characters. The array has three elements. Initially they are a pointer to the string `"Venus"`, a pointer to `"Jupiter"`, and a pointer to `"Saturn"`:

```
static char *planets[ ] = { "Venus", "Jupiter", "Saturn" };
```

The wchar_t Type Specifier: The **wchar_t** type specifier is an integral type that has enough storage to represent a wide character literal. (A wide character literal is a character literal that is prefixed with the letter `L`, for example `L'x'`)

Floating-Point Variables

There are three types of floating-point variables:

- **float**
- **double**
- **long double**

To declare a data object that is a floating-point type, use the following **float** specifier:

```
►►──┬─float───────┬──────────────────────────────────────────────────►◄
    ├─double──────┤
    └─long double─┘
```

The declarator for a simple floating-point declaration is an identifier. Initialize a simple floating-point variable with a float constant or with a variable or expression that evaluates to an integer or floating-point number. The storage class of a variable determines how you initialize the variable.

Examples of Floating-Point Data Types

The following example defines the identifier `pi` as an object of type **double**:

```
double pi;
```

The following example defines the **float** variable `real_number` with the initial value `100.55`:

```
static float real_number = 100.55f;
```

Note: If you do not add the **f** suffix to a floating-point literal, that number will be of type **double**. If you initialize an object of type **float** with an object of type **double**, the compiler will implicitly convert the object of type **double** to an object of type **float**.

The following example defines the **float** variable `float_var` with the initial value `0.0143`:

```
float float_var = 1.43e-2f;
```

The following example declares the **long double** variable `maximum`:

```
extern long double maximum;
```

The following example defines the array `table` with 20 elements of type **double**:

```
double table[20];
```

Related References
- “Floating-Point Literals” on page 23
- “Assignment Expressions” on page 144

Integer Variables

Integer variables fall into the following categories:
- integral types
  - C++ **bool**
  - **char**
  - **wchar_t**
  - signed integer types
    - **signed char**
    - **short int**
    - **int**
    - **long int**
    - **long long int**
  - unsigned integer types
    - **unsigned char**
    - **unsigned short int**
    - **unsigned int**
    - **unsigned long int**
    - **unsigned long long int**

Note: C++ The integer types **long long int** and **unsigned long long int** are orthogonal extensions to Standard C++.

The default integer type for a bit field is **unsigned**.

The amount of storage allocated for integer data is implementation-dependent.

The **unsigned** prefix indicates that the object is a nonnegative integer. Each unsigned type provides the same size storage as its signed equivalent. For example, **int** reserves the same storage as **unsigned int**. Because a signed type reserves a sign bit, an unsigned type can hold a larger positive integer value than the equivalent signed type.

The declarator for a simple integer definition or declaration is an identifier. You can initialize a simple integer definition with an integer constant or with an expression that evaluates to a value that can be assigned to an integer. The storage class of a variable determines how you can initialize the variable.

C++ When the arguments in overloaded functions and overloaded operators are integer types, two integer types that both come from the same group are not treated as distinct types. For example, you cannot overload an **int** argument against a **signed int** argument.

Examples of Integer Data Types

The following example defines the **short int** variable `flag`:

```
short int flag;
```

The following example defines the **int** variable `result`:

```
int result;
```

The following example defines the **unsigned long int** variable `ss_number` as having the initial value `438888834` :

```
unsigned long ss_number = 438888834ul;
```

void Type

The **void** data type always represents an empty set of values. The only object that can be declared with the type specifier **void** is a pointer.

When a function does not return a value, you should use **void** as the type specifier in the function definition and declaration. An argument list for a function taking no arguments is **void**.

You cannot declare a variable of type **void**, but you can explicitly convert any expression to type **void**. The resulting expression can only be used as one of the following:
- An expression statement
- The left operand of a comma expression
- The second or third operand in a conditional expression.

Example of void Type

In the following example, the function `find_max` is declared as having type **void**.

Note: C The use of the `sizeof` operator in the line `find_max(numbers, (sizeof(numbers) / sizeof(numbers[0])));` is a standard method of determining the number of elements in an array.

```
/**
** Example of void type
**/
#include <stdio.h>

/* declaration of function find_max */
extern void find_max(int x[ ], int j);

int main(void)
{
   static int numbers[ ] = { 99, 54, -102, 89};

   find_max(numbers, (sizeof(numbers) / sizeof(numbers[0])));

   return(0);
}

void find_max(int x[ ], int j)
{ /* begin definition of function find_max */
   int i, temp = x[0];

   for (i = 1; i < j; i++)
   {
       if (x[i] > temp)
          temp = x[i];
   }
   printf("max number = %d\n", temp);
} /* end definition of function find_max  */
```

Compound Types

C++ C++ formally defines the concept of a compound type and how one can be constructed. Many of the compound types originated in C.

You are using a compound type when you construct any of the following:

- An array of objects of a given type
- Any functions, which have parameters of a given type and return void or objects of a given type
- A pointer to **void**, to an object, or to a function of a given type
- A reference to an object or function of a given type
- A class
- A union
- An enumeration
- A pointer to a non-static class member

Structures

A *structure* contains an ordered group of data objects. Unlike the elements of an array, the data objects within a structure can have varied data types. Each data object in a structure is a *member* or *field*.

C A member of a structure may have any object type other than a variably modified type. Every member except the last must be a complete type. As a special case, the last element of a structure with more than one member may have an incomplete array type, which is called a *flexible array member*.

C++ In C++, a structure member must be a complete type.

Use structures to group logically related objects. For example, to allocate storage for the components of one address, define the following variables:

```
int street_no;
char *street_name;
char *city;
char *prov;
char *postal_code;
```

To allocate storage for more than one address, group the components of each address by defining a structure data type and as many variables as you need to have the structure data type.

C++ In C++, a structure is the same as a class except that its members and inheritance are public by default.

In the following example, line int street_no; through to char *postal_code; declare the structure tag address:

```
struct address {
                 int street_no;
                 char *street_name;
                 char *city;
                 char *prov;
                 char *postal_code;
               };
struct address perm_address;
struct address temp_address;
struct address *p_perm_address = &perm_address;
```

The variables perm_address and temp_address are instances of the structure data type address. Both contain the members described in the declaration of address. The pointer p_perm_address points to a structure of address and is initialized to point to perm_address.

Refer to a member of a structure by specifying the structure variable name with the dot operator (.) or a pointer with the arrow operator (->) and the member name. For example, both of the following:

```
perm_address.prov = "Ontario";
p_perm_address -> prov = "Ontario";
```

assign a pointer to the string "Ontario" to the pointer prov that is in the structure perm_address.

All references to structures must be fully qualified. In the example, you cannot reference the fourth field by prov alone. You must reference this field by perm_address.prov.

Structures with identical members but different names are not compatible and cannot be assigned to each other.

Structures are not intended to conserve storage. If you need direct control of byte mapping, use pointers.

Compatible Structures

C Each structure definition creates a new structure type that is neither the same as nor compatible with any other structure type in the same source file. However, a type specifier that is a reference to a previously defined structure type is the same type. The structure tag associates the reference with the definition, and effectively acts as the type name. To illustrate this, only the types of structures j and k are the same.

```
struct   { int a; int b; } h;
struct   { int a; int b; } i;
struct S { int a; int b; } j;
struct S k;
```

Declaring and Defining a Structure: A *structure type definition* describes the members that are part of the structure. It contains the **struct** keyword followed by an optional identifier (the structure tag) and a brace-enclosed list of members.

A declaration of a structure data type has the form:

```
►►──struct──┬─identifier──────────────────────────────┬──────────────────►◄
            │                      ┌───────────┐      │
            │                      ▼           │      │
            └─┬────────────┬──{───member──;───┴──}──┘
              └─identifier─┘
```

The keyword **struct** followed by an identifier (tag) gives a name to the data type. If you do not provide a tag name, you must put all variable definitions that refer to it within the declaration of the data type.

A *structure declaration* has the same form as a structure definition except the declaration does not have a brace-enclosed list of members. A *structure definition has the same form as the declaration of that structure data type, but ends with a semicolon.*

Defining Structure Members

The list of members provides the structure data type with a description of the values that can be stored in the structure. In C, a structure member may be of any type except "function returning `T`" (for some type `T`), any incomplete type, any variably modified type, and `void`. Because incomplete types are not allowed as a structure member, a structure type may not contain an instance of itself as a member, but is allowed to contain a pointer to an instance of itself.

The definition of a structure member has the form of a variable declaration. The names of structure members must be distinct within a single structure, but the same member name may be used in another structure type that is defined within the same scope, and may even be the same as a variable, function, or type name. A member that does not represent a bit field can be of any data type, which can be qualified with either of the type qualifiers **volatile** or **const**. The result is an lvalue. However, a bit field without a type qualifier can be declared as a structure member. If the bit field is unnamed, it does not participate in initialization, and will have indeterminate value after initialization.

To allow proper alignment of components, holes or padding may appear between any consecutive members in the structure layout.

Flexible Array Members

The last element of a structure with more than one named member may be an incomplete array type, referred to as a *flexible array member*.

A *flexible array member* is an element of a structure with more than one named member. The flexible array member must be the last element of such a structure and must be of an incomplete array type.

```
►►──array_identifier[ ]──────────────────────────────────────────────►◄
```

For example, `b` is a flexible array member of `struct foo`.

```
struct foo{
   int a;
   char b[];
};
```

The size of `struct foo` is 4. `struct foo` cannot be a member of another struct or array.

When the array subscript is zero, the array member is considered a *zero-extent array*.

```
►►—array_identifier[0]————————————————————————————————————————►◄
```

If in the previous example b is declared as a zero-extent array, the size of struct foo is still 4, but struct foo is allowed to be a member of another struct or array, as in the following example.

```
struct bar{
   struct foo zearray;
};
```

Usually a flexible array member is ignored. However, it is recognized in two cases:
- Suppose that an array of unspecified length replaces the flexible array member. The flexible array member of the original structure is recognized when the size of the original structure is equal to the offset of the last element of the structure with the replacement array.
- When the dot or arrow operator is used to represent the flexible array member.

In the second case, the behavior is as if that member were replaced with the longest array that would not make the structure larger than the object being accessed. The offset of the array remains the same as that of the flexible array member. If the replacement array would have no elements, the behavior is as if it had one element, but that element may not be accessed, nor can a pointer one past it be generated. To illustrate, d is the flexible array member of the structure struct s.

```
// Assuming the same alignment for all array members,

struct s { int n; double d[]; };
struct ss { int n; double d[1]; };
```

The expressions offsetof(struct s, d) and offsetof(struct ss, d) have the same value: sizeof(struct s).

Defining a Structure Variable: ► C A structure variable definition contains an optional storage class keyword, the **struct** keyword, a structure tag, a declarator, and an optional identifier. The structure tag indicates the data type of the structure variable.

► C++ The keyword **struct** is optional in C++.

You can declare structures having any storage class. Structures declared with the **register** storage class specifier are treated as automatic structures.

Initializing Structures: An initializer for a structure is a brace-enclosed comma-separated list of values. An initializer is preceded by an equal sign (=). In the absence of designations, memory for structure members is allocated in the order declared, and memory address are assigned in increasing order, with the first component starting at the beginning address of the structure name itself. You do not have to initialize all members of a structure. The default initializer for a structure with static storage is the recursive default for each component; a structure with automatic storage has none.

► C The following subsection pertains to C only.

Named members of a structure can be initialized in any order; any named member of a union can be initialized, even if it is not the first member. A *designator* identifies the structure or union member to be initialized. The designator for a structure or union member consists of a dot and its identifier (*.fieldname*). A *designator list* is a combination of one or more designators for any of the aggregate types. A *designation* is a designator list followed by an equal sign (=).

A designator identifies a first subobject of the current object, which at the beginning of the initialization is the structure itself. After initializing the first subobject, the next subobject becomes the current object, and its first subobject is initialized; that is, initialization proceeds in forward order, and any previous subobject initializations are overridden.

The initializer for an automatic variable of a structure or any aggregate type can be a constant or non-constant expression. Allowing an initializer to be a constant or non-constant expression is a C99 language feature.

The following declaration of a structure is a definition that contains designators, which remove some of the ambiguity about which subobject will be initialized by providing an explicit initialization. The following declaration defines an array with two element structures. In the excerpt below, `[0].a` and `[1].a[0]` are designator lists.

```
struct { int a[5], b; } game[] =
        { [0].a = { 1 }, [1].a[0] = 2 };

   /* game[0].a[0] is 1, game[1].a[0] is 2, and all other elements are zero. */
```

The declaration syntax uses braces to indicate initializer lists, yet is referred to as a *bracketed form*. A fully bracketed form of a declaration is less likely to be misunderstood than a terser form. The following definition accomplishes the same thing, is legal and shorter, but inconsistently bracketed, and could be misleading. Neither `b` structure member of the two `struct game` objects is initialized to 2.

```
struct { int a[5], b; } game[] =
        { { 1 }, 2 };

   /* game[0].a[0] is 1, game[1].a[0] is 2, and all other elements are zero. */
```

Unnamed structure or union members do not participate in initialization and have indeterminate value after initialization.

Example

The following definition shows a completely initialized structure:

```
struct address {
               int street_no;
               char *street_name;
               char *city;
               char *prov;
               char *postal_code;
             };
static struct address perm_address =
             { 3, "Savona Dr.", "Dundas", "Ontario", "L4B 2A1"};
```

The values of `perm_address` are:

Member	**Value**
`perm_address.street_no`	3
`perm_address.street_name`	address of string "`Savona Dr.`"
`perm_address.city`	address of string "`Dundas`"
`perm_address.prov`	address of string "`Ontario`"

perm_address.postal_code	address of string "L4B 2A1"

The following definition shows a partially initialized structure:

```
struct address {
                 int street_no;
                 char *street_name;
                 char *city;
                 char *prov;
                 char *postal_code;
               };
struct address temp_address =
               { 44, "Knyvet Ave.", "Hamilton", "Ontario" };
```

The values of temp_address are:

Member	**Value**
temp_address.street_no	44
temp_address.street_name	address of string "Knyvet Ave."
temp_address.city	address of string "Hamilton"
temp_address.prov	address of string "Ontario"
temp_address.postal_code	value depends on the storage class.

Note: The initial value of uninitialized structure members like temp_address.postal_code depends on the storage class associated with the member.

Declaring Structure Types and Variables in the Same Statement: C To define a structure type and a structure variable in one statement, put a declarator and an optional initializer after the type definition. To specify a storage class specifier for the variable, you must put the storage class specifier at the beginning of the statement.

For example:

```
static struct {
                 int street_no;
                 char *street_name;
                 char *city;
                 char *prov;
                 char *postal_code;
              } perm_address, temp_address;
```

Because this example does not name the structure data type, perm_address and temp_address are the only structure variables that will have this data type. Putting an identifier after **struct**, lets you make additional variable definitions of this data type later in the program.

The structure type (or tag) cannot have the **volatile** qualifier, but a member or a structure variable can be defined as having the **volatile** qualifier.

For example:

```
static struct class1 {
                        char descript[20];
                        volatile long code;
                        short complete;
                     } volatile file1, file2;
struct class1 subfile;
```

This example qualifies the structures file1 and file2, and the structure member subfile.code as **volatile**.

Alignment of Structures: Linux Structures are aligned according to the setting of the `align` compiler option, which specifies the alignment rules the compiler is to use when laying out the storage of structures and unions. Each of the suboptions affects the alignment in a different way. The mapping of a structure is based on the setting in effect at the end of the structure definition. Structure members are aligned by type.

A `#pragma options align` directive can be embedded within a structure or union definition. The alignment will apply only to definitions of nested structures or unions, and to any structures or unions that appear after the closing brace of the outer structure or union. The alignment of the outer structure or union will be that which was in effect at the opening brace of its definition.

Structures and unions with different alignments can be nested. Each structure is laid out using the alignment applicable to it. The start position of the nested structure is determined by the alignment of the structure in which it is nested.

Structures and unions with identical members but using different alignments are not type-compatible and cannot be assigned to each other.

Related References
- For a full discussion of the `align` compiler option and the `#pragmas` affecting alignment, see *XL C/C++ Compiler Reference*: Data Mapping and Storage

Declaring and Using Bit Fields in Structures: Both C and C++ allow integer members to be stored into memory spaces smaller than the compiler would ordinarily allow. These space-saving structure members are called *bit fields*, and their width in bits can be explicitly declared. Bit fields are used in programs that must force a data structure to correspond to a fixed hardware representation and are unlikely to be portable.

The syntax for declaring a bit field is as follows:

```
►►──type_specifier──┬────────────┬──:──constant_expression──;──────────────►◄
                    └─declarator─┘
```

A bit field declaration contains a type specifier followed by an optional declarator, a colon, a constant integer expression that indicates the field width in bits, and a semicolon. A bit field declaration may not use either of the type qualifiers, **const** or **volatile**.

C The C99 standard requires the allowable data types for a bit field to include qualified and unqualified **_Bool**, **signed int**, and **unsigned int**. In addition, this implementation supports the following types.
- **int**
- **short**, **signed short**, **unsigned short**
- **char**, **signed char**, **unsigned char**
- **long**, **signed long**, **unsigned long**
- **long long**, **signed long long**, **unsigned long long**

In all implementations, the default integer type for a bit field is **unsigned**.

C++ C++ extends the list of allowable types for bit fields to include any integral type or enumeration type.

In either language, when you assign a value that is out of range to a bit field, the low-order bit pattern is preserved and the appropriate bits are assigned.

Bit fields with a length of 0 must be unnamed. Unnamed bit fields cannot be referenced or initialized. A zero-width bit field can cause the next field to be aligned on the next container boundary where the container is the same size as the underlying type of the bit field.

Linux Bit fields are also subject to the align compiler option. Each of the align suboptions gives a different set of alignment properties to the bit fields. For a full discussion of the align compiler option and the #pragmas affecting alignment, see *XL C/C++ Compiler Reference*.

Linux The maximum bit-field length is 64 bits. For portability, do not use bit fields greater than 32 bits in size.

The following restrictions apply to bit fields. You cannot:
- Define an array of bit fields
- Take the address of a bit field
- Have a pointer to a bit field
- Have a reference to a bit field

The following structure has three bit-field members kingdom, phylum, and genus, occupying 12, 6, and 2 bits respectively:

```
struct taxonomy {
     int kingdom : 12;
     int phylum : 6;
     int genus : 2;
     };
```

Alignment of Bit Fields

If a series of bit fields does not add up to the size of an **int**, padding can take place. The amount of padding is determined by the alignment characteristics of the members of the structure.

The following example demonstrates padding, and is valid for all implementations. Suppose that an **int** occupies 4 bytes. The example declares the identifier kitchen to be of type struct on_off:

```
struct on_off {
                  unsigned light : 1;
                  unsigned toaster : 1;
                  int count;            /* 4 bytes */
                  unsigned ac : 4;
                  unsigned : 4;
                  unsigned clock : 1;
                  unsigned : 0;
                  unsigned flag : 1;
                 } kitchen ;
```

The structure kitchen contains eight members totalling 16 bytes. The following table describes the storage that each member occupies:

Member Name	Storage Occupied
light	1 bit
toaster	1 bit

Member Name	Storage Occupied
(padding — 30 bits)	To the next **int** boundary
count	The size of an **int** (4 bytes)
ac	4 bits
(unnamed field)	4 bits
clock	1 bit
(padding — 23 bits)	To the next **int** boundary (unnamed field)
flag	1 bit
(padding — 31 bits)	To the next **int** boundary

All references to structure fields must be fully qualified. For instance, you cannot reference the second field by toaster. You must reference this field by kitchen.toaster.

The following expression sets the light field to 1:

```
kitchen.light = 1;
```

When you assign to a bit field a value that is out of its range, the bit pattern is preserved and the appropriate bits are assigned. The following expression sets the toaster field of the kitchen structure to 0 because only the least significant bit is assigned to the toaster field:

```
kitchen.toaster = 2;
```

Example Program Using Structures: The following program finds the sum of the integer numbers in a linked list:

```
/**
 ** Example program illustrating structures using linked lists
 **/

#include <stdio.h>

struct record {
                int number;
                struct record *next_num;
              };

int main(void)
{
   struct  record name1, name2, name3;
   struct  record *recd_pointer = &name1;
   int sum = 0;

   name1.number = 144;
   name2.number = 203;
   name3.number = 488;

   name1.next_num = &name2;
   name2.next_num = &name3;
   name3.next_num = NULL;

   while (recd_pointer != NULL)
   {
      sum += recd_pointer->number;
      recd_pointer = recd_pointer->next_num;
   }
   printf("Sum = %d\n", sum);

   return(0);
}
```

The structure type `record` contains two members: the integer `number` and `next_num`, which is a pointer to a structure variable of type `record`.

The `record` type variables `name1`, `name2`, and `name3` are assigned the following values:

Member Name	**Value**
`name1.number`	`144`
`name1.next_num`	The address of `name2`
`name2.number`	`203`
`name2.next_num`	The address of `name3`
`name3.number`	`488`
`name3.next_num`	`NULL` (Indicating the end of the linked list.)

The variable `recd_pointer` is a pointer to a structure of type `record`. It is initialized to the address of `name1` (the beginning of the linked list).

The **while** loop causes the linked list to be scanned until `recd_pointer` equals `NULL`. The statement:

```
recd_pointer = recd_pointer->next_num;
```

advances the pointer to the next object in the list.

Related References
- "Incomplete Types" on page 77

Unions

A *union* is an object similar to a structure except that all of its members start at the same location in memory. A union can contain the value of only one of its members at a time. The default initializer for a union with `static` storage is the default for the first component; a union with `automatic` storage has none.

The storage allocated for a union is the storage required for the largest member of the union (plus any padding that is required so that the union will end at a natural boundary of its member having the most stringent requirements). For this reason, variably modified types may not be declared as union members. All of a union's components are effectively overlaid in memory: each member of a union is allocated storage starting at the beginning of the union, and only one member can occupy the storage at a time.

▶ C Any member of a union can be initialized, not just the first member, by using a designator. A designated initializer for a union has the same syntax as that for a structure. In the following example, the designator is `.any_member` and the initializer is `{.any_member = 13 }`:

```
union { /* ... */ } caw = { .any_member = 13 };
```

Compatible Unions

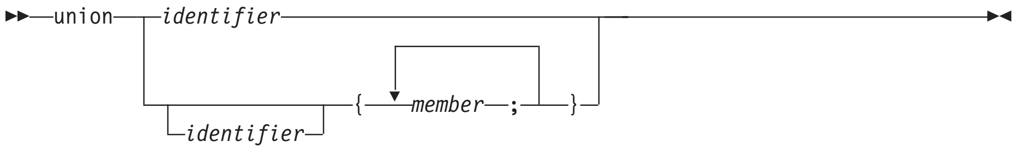
C Each union definition creates a new union type that is neither the same as nor compatible with any other union type in the same source file. However, a type specifier that is a reference to a previously defined union type is the same type. The union tag associates the reference with the definition, and effectively acts as the type name.

C++ In C++, a union is a limited form of the class type. It can contain access specifiers (public, protected, private), member data, and member functions, including constructors and destructors. It cannot contain virtual member functions or static data members. Default access of members in a union is public. A union cannot be used as a base class and cannot be derived from a base class.

C++ places additional limitations on the allowable data types for a union member. In C++, a union member cannot be a class object that has a constructor, destructor, or overloaded copy assignment operator, nor can it be of reference type. A union member cannot be declared with the keyword **static**.

Declaring a Union: A *union type definition* contains the **union** keyword followed by an optional identifier (tag) and a brace-enclosed list of members.

A union definition has the following form:



A *union declaration* has the same form as a union definition except that the declaration has no brace-enclosed list of members.

The *identifier* is a tag given to the union specified by the member list. Once a tag is specified, any subsequent declaration of the union (in the same scope) can be made by declaring the tag and omitting the member list. If a tag is not specified, all variable definitions that refer to that union must be placed within the statement that defines the data type.

The list of members provides the data type with a description of the objects that can be stored in the union.

A union member definition has same form as a variable declaration.

A member of a union can be referenced the same way as a member of a structure.

For example:

```
union {
      char birthday[9];
      int age;
      float weight;
      } people;

people.birthday[0] = '\n';
```

assigns `'\n'` to the first element in the character array `birthday`, a member of the union `people`.

A union can represent only one of its members at a time. In the example, the union people contains either age, birthday, or weight but never more than one of these. The printf statement in the following example does not give the correct result because people.age replaces the value assigned to people.birthday in the first line:

```
#include <stdio.h>
#include <string.h>

union {
  char birthday[9];
  int age;
  float weight;
} people;

int main(void) {
  strcpy(people.birthday, "03/06/56");
  printf("%s\n", people.birthday);
  people.age = 38;
  printf("%s\n", people.birthday);
}
```

The output of the above example will be similar to the following:

```
03/06/56
&
```

Defining a Union Variable: ▶ C A union variable definition has the following form:

```
►►──┬────────────────────────┬──union──union_data_type_name──identifier──────────►
    └─storage_class_specifier─┘

►──┬────────────────────────┬──────────────────────────────────────────────────►◄
   └─=──initialization_value─┘
```

You must declare the union data type before you can define a union having that type.

Any named member of a union can be initialized, even if it is not the first member. The initializer for an automatic variable of union type can be a constant or non-constant expression. Allowing a nonconstant aggregate initializer is a C99 language feature.

The following example shows how you would initialize the first union member birthday of the union variable people:

```
union {
      char birthday[9];
      int age;
      float weight;
      } people = {"23/07/57"};
```

You can define a union data type and a union of that type in the same statement by placing the variable declarator after the data type definition. The storage class specifier for the variable must appear at the beginning of the statement.

Anonymous Unions: An *anonymous union* is a union without a class name. It cannot be followed by a declarator. An anonymous union is not a type; it defines an unnamed object and it cannot have member functions.

The member names of an anonymous union must be distinct from other names within the scope in which the union is declared. You can use member names directly in the union scope without any additional member access syntax.

For example, in the following code fragment, you can access the data members `i` and `cptr` directly because they are in the scope containing the anonymous union. Because `i` and `cptr` are union members and have the same address, you should only use one of them at a time. The assignment to the member `cptr` will change the value of the member `i`.

```
void f()
{
union { int i; char* cptr ; };
/* . . .  */
i = 5;
cptr = "string_in_union"; // overrides the value 5
}
```

C++ An anonymous union cannot have protected or private members. A global or namespace anonymous union must be declared with the keyword **static**.

Examples of Unions: The following example defines a union data type (not named) and a union variable (named `length`). The member of `length` can be a **long int**, a **float**, or a **double**.

```
union {
        float meters;
        double centimeters;
        long inches;
      } length;
```

The following example defines the union type `data` as containing one member. The member can be named `charctr`, `whole`, or `real`. The second statement defines two `data` type variables: `input` and `output`.

```
union data {
             char charctr;
             int whole;
             float real;
           };
union data input, output;
```

The following statement assigns a character to `input`:

```
input.charctr = 'h';
```

The following statement assigns a floating-point number to member `output`:

```
output.real = 9.2;
```

The following example defines an array of structures named `records`. Each element of `records` contains three members: the integer `id_num`, the integer `type_of_input`, and the `union` variable `input`. `input` has the `union` data type defined in the previous example.

```
struct {
         int id_num;
         int type_of_input;
         union data input;
       } records[10];
```

The following statement assigns a character to the structure member `input` of the first element of **records**:

```
records[0].input.charctr = 'g';
```

Enumerations

An *enumeration* is a data type consisting of a set of values that are named integral constants. It is also referred to as an *enumerated type* because you must list (enumerate) each of the values in creating a name for each of them. A named value in an enumeration is called an *enumeration constant*. In addition to providing a way of defining and grouping sets of integral constants, enumerations are useful for variables that have a small number of possible values.

You can define an enumeration data type and all variables that have that enumeration type in one statement, or you can declare an enumeration type separately from the definition of variables of that type. The identifier associated with the data type (not an object) is called an *enumeration tag*. Each distinct enumeration is a different enumeration type.

Compatible Enumerations

C In C, each enumerated type must be compatible with the integer type that represents it. Enumeration variables and constants are treated by the compiler as integer types. Consequently, in C you can freely mix the values of different enumerated types, regardless of type compatibility.

Linux C Compatibility between an enumerated type and the integer type that represents it is controlled by compiler options and related pragmas. For a full discussion of the enum compiler option and related #pragmas, see *XL C/C++ Compiler Reference*

C++ C++ treats enumerated types as distinct from each other and from integer types. You must explicitly cast an integer in order to use it as an enumeration value.

Declaring an Enumeration Data Type: An enumeration type declaration contains the **enum** keyword followed by an optional identifier (the enumeration tag) and a brace-enclosed list of enumerators. Commas separate each enumerator in the enumerator list. C99 allows a trailing comma between the last enumerator and the closing brace. A declaration of an enumeration has the form:

```
                                 ┌─,──────────┐
                                 ↓            │
►►──enum──┬────────────┬──{──────enumerator───┴──}──;──────────────────►◄
          └─identifier─┘
```

The keyword **enum**, followed by the identifier, names the data type (like the tag on a **struct** data type). The list of enumerators provides the data type with a set of values.

In C, each enumerator represents an integer value. In C++, each enumerator represents a value that can be converted to an integral value.

An enumerator has the form:

```
►►──identifier──┬──────────────────────────────┬───────────────────────►◄
                └─=──integral_constant_expression─┘
```

To conserve space, enumerations may be stored in spaces smaller than that of an **int**.

Enumeration Constants: When you define an enumeration data type, you specify a set of identifiers that the data type represents. Each identifier in this set is an *enumeration constant*.

The value of the constant is determined in the following way:

1. An equal sign (=) and a constant expression after the enumeration constant gives an explicit value to the constant. The identifier represents the value of the constant expression.
2. If no explicit value is assigned, the leftmost constant in the list receives the value zero (0).
3. Identifiers with no explicitly assigned values receive the integer value that is one greater than the value represented by the previous identifier.

▶ C In C, enumeration constants have type **int**. If a constant expression is used as an initializer, the value of the expression cannot exceed the range of **int** (that is, INT_MIN to INT_MAX as defined in the header <limits.h>).

▶ C++ In C++, each enumeration constant has a value that can be promoted to a signed or unsigned integer value and a distinct type that does not have to be integral. Use an enumeration constant anywhere an integer constant is allowed, or for C++, anywhere a value of the enumeration type is allowed.

Each enumeration constant must be unique within the scope in which the enumeration is defined. In the following example, second declarations of average and poor cause compiler errors:

```
func()
    {
       enum score { poor, average, good };
       enum rating { below, average, above };
       int poor;
    }
```

The following data type declarations list oats, wheat, barley, corn, and rice as enumeration constants. The number under each constant shows the integer value.

```
enum grain { oats, wheat, barley, corn, rice };
   /*         0      1      2      3     4         */

enum grain { oats=1, wheat, barley, corn, rice };
   /*         1        2      3      4     5       */

enum grain { oats, wheat=10, barley, corn=20, rice };
   /*          0     10        11     20       21  */
```

It is possible to associate the same integer with two different enumeration constants. For example, the following definition is valid. The identifiers suspend and hold have the same integer value.

```
enum status { run, clear=5, suspend, resume, hold=6 };
   /*          0      5        6       7       6       */
```

Defining Enumeration Variables: An enumeration variable definition has the following form:

```
►►──┬─────────────────────────┬──enum──enumeration_data_type_name──identifier──────►
    └─storage_class_specifier─┘

►──┬──────────────────────────┬────────────────────────────────────────────────────►◄
   └─=──enumeration_constant──┘
```

You must declare the enumeration data type before you can define a variable having that type.

C++ The initializer for an enumeration variable contains the = symbol followed by an expression *enumeration_constant*. In C++, the initializer must have the same type as the associated enumeration type.

The first line of the following example declares the enumeration `grain`. The second line defines the variable `g_food` and gives `g_food` the initial value of `barley` (2).

```
enum grain { oats, wheat, barley, corn, rice };
enum grain g_food = barley;
```

The type specifier `enum grain` indicates that the value of `g_food` is a member of the enumerated data type `grain`.

C++ The **enum** keyword is optional when declaring a variable with enumeration type. However, it is required when declaring the enumeration itself. For example, both statements declare a variable of enumeration type:

```
enum grain g_food = barley;
     grain cob_food = corn;
```

Defining an Enumeration Type and Enumeration Objects: You can define a type and a variable in one statement by using a declarator and an optional initializer after the type definition. To specify a storage class specifier for the variable, you must put the storage class specifier at the beginning of the declaration. For example:

```
register enum score { poor=1, average, good } rating = good;
```

C++ C++ also lets you put the storage class immediately before the declarator list. For example:

```
enum score { poor=1, average, good } register rating = good;
```

Either of these examples is equivalent to the following two declarations:

```
enum score { poor=1, average, good };
register enum score rating = good;
```

Both examples define the enumeration data type `score` and the variable `rating`. `rating` has the storage class specifier **register**, the data type `enum score`, and the initial value `good`.

Combining a data type definition with the definitions of all variables having that data type lets you leave the data type unnamed. For example:

```
enum { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
      Saturday } weekday;
```

defines the variable `weekday`, which can be assigned any of the specified enumeration constants.

Example Program Using Enumerations: The following program receives an integer as input. The output is a sentence that gives the French name for the weekday that is associated with the integer. If the integer is not associated with a weekday, the program prints "C'est le mauvais jour."

```
/**
 ** Example program using enumerations
 **/

#include <stdio.h>

enum days {
            Monday=1, Tuesday, Wednesday,
            Thursday, Friday, Saturday, Sunday
          } weekday;

void french(enum days);

int main(void)
{
   int num;

   printf("Enter an integer for the day of the week.  "
          "Mon=1,...,Sun=7\n");
   scanf("%d", &num);
   weekday=num;
   french(weekday);
   return(0);
}
void french(enum days weekday)
{
   switch (weekday)
   {
      case Monday:
         printf("Le jour de la semaine est lundi.\n");
         break;
      case Tuesday:
         printf("Le jour de la semaine est mardi.\n");
         break;
      case Wednesday:
         printf("Le jour de la semaine est mercredi.\n");
         break;
      case Thursday:
         printf("Le jour de la semaine est jeudi.\n");
         break;
      case Friday:
         printf("Le jour de la semaine est vendredi.\n");
         break;
      case Saturday:
         printf("Le jour de la semaine est samedi.\n");
         break;
      case Sunday:
         printf("Le jour de la semaine est dimanche.\n");
         break;
      default:
         printf("C'est le mauvais jour.\n");
   }
}
```

Complex Types

Complex types consist of two parts: a real part and an imaginary part. Imaginary types consist of only the imaginary part.

There are three type specifiers for complex types:
- **float**
- **double**
- **long double**

To declare a data object that is a complex type, use the one of the following type specifiers:

```
►►──┬─float───────┬──complex──────────────────────────────────────────────────►◄
    ├─double──────┤
    └─long double─┘
```

The imaginary unit I is a constant of type **float complex**. The predefined macro `_Complex_I` represents a constant expression of type `const float _Complex`, with the value of the imaginary unit.

The complex type and the real floating type are collectively called the *floating types*. Each floating type has a corresponding real type. For a real floating type, it is the same type. For a complex type, it is the type given by deleting the keyword **_Complex** from the type name.

The representation and alignment requirements of a complex type are the same as an array type containing two elements of the corresponding real type. The real part is equal to the first element; the imaginary part is equal to the second element.

Arithmetic conversions are the same as those for the real type of the complex type. If either operand is a complex type, the result is a complex type, and the operand having the smaller type for its real part is promoted to the complex type corresponding to the larger of the real types. For example, a **double _Complex** added to a **float _Complex** will yield a result of type **double _Complex**.

When casting a complex type to a real type, the imaginary part is dropped. When the value of a real type is converted to a complex type, the real part of the complex result value is determined by the rules of conversion to the corresponding real type, and the imaginary part of the complex result value is a positive zero or an unsigned zero.

The equality and inequality operators have the same behavior as for real types. None of the relational operators may have a complex type as an operand.

Related References
- "Complex Literals" on page 26

Type Qualifiers

C recognizes three type qualifiers, **const**, **volatile**, and **restrict**. C++ refers to the type qualifiers **const** and **volatile** as *cv-qualifiers* and recognizes the type qualifier **restrict** as a language extension. In both languages, the cv-qualifiers are only meaningful in expressions that are lvalues. C++ allows a cv-qualifier to apply to functions, which is disallowed in C. The type qualifier **restrict** may only be applied to pointers.

Syntax for the const and volatile keywords

For a **volatile** or **const** pointer, you must put the keyword between the * and the identifier. For example:

```
int * volatile x;        /* x is a volatile pointer to an int */
int * const y = &z;      /* y is a const pointer to the int variable z */
```

For a pointer to a **volatile** or **const** data object, the type specifier, qualifier, and storage class specifier can be in any order. For example:

```
volatile int *x;         /* x is a pointer to a volatile int
   or                                                        */
int volatile *x;         /* x is a pointer to a volatile int  */

const int *y;            /* y is a pointer to a const int
   or                                                        */
int const *y;            /* y is a pointer to a const int     */
```

In the following example, the pointer to y is a constant. You can change the value that y points to, but you cannot change the value of y:

```
int * const y
```

In the following example, the value that y points to is a constant integer and cannot be changed. However, you can change the value of y:

```
const int * y
```

For other types of **volatile** and **const** variables, the position of the keyword within the definition (or declaration) is less important. For example:

```
volatile struct omega {
                         int limit;
                         char code;
                      } group;
```

provides the same storage as:

```
struct omega {
                int limit;
                char code;
             } volatile group;
```

In both examples, only the structure variable group receives the **volatile** qualifier. Similarly, if you specified the **const** keyword instead of **volatile**, only the structure variable group receives the **const** qualifier. The **const** and **volatile** qualifiers when applied to a structure, union, or class also apply to the members of the structure, union, or class.

Although enumeration, class, structure, and union variables can receive the **volatile** or **const** qualifier, enumeration, class, structure, and union tags do not carry the **volatile** or **const** qualifier. For example, the blue structure does not carry the **volatile** qualifier:

```
volatile struct whale {
                     int weight;
                     char name[8];
                } beluga;
struct whale blue;
```

The keywords **volatile** and **const** cannot separate the keywords **enum**, **class**, **struct**, and **union** from their tags.

You can declare or define a **volatile** or **const** function only if it is a nonstatic member function. You can define or declare any function to return a pointer to a **volatile** or **const** function.

An item can be both **const** and **volatile**. In this case the item cannot be legitimately modified by its own program but can be modified by some asynchronous process.

You can put more than one qualifier on a declaration: the compiler ignores duplicate type qualifiers.

The const Type Qualifier

C The **const** qualifier explicitly declares a data object as something that cannot be changed. Its value is set at initialization. You cannot use **const** data objects in expressions requiring a modifiable lvalue. For example, a **const** data object cannot appear on the lefthand side of an assignment statement.

An object that is declared **const** is guaranteed to remain constant for its lifetime, not throughout the entire execution of the program. For this reason, a `const` object cannot be used in constant expressions. In the following example, the `const` object `k` is declared within `foo`, is initialized to the value of `foo`'s argument, and remains constant until the function returns. In C, `k` cannot be used to specify the length of an array because that value will not be known until `foo` is called.

```
void foo(int j)
{
   const int k = j;
   int ary[k];     /* Violates rule that the length of each
                      array must be known to the compiler   */
}
```

In C, a `const` object that is declared outside a block has external linkage and can be shared among files. In the following example, you cannot use `k` to specify the length of the array because it is probably defined in another file.

```
extern const int k;
int ary[k];     /* Another violation of the rule that the length of
                   each array must be known to the compiler   */
```

A top-level declaration of a `const` object without an explicit storage class is considered to be **extern** in C, but is considered **static** in C++.

```
const int k = 12;  /* Different meanings in C and C++ */

static const int k2 = 120;  /* Same meaning in C and C++ */
extern const int k3 = 121;  /* Same meaning in C and C++ */
```

In C++, all `const` declarations must have initializers, except those referencing externally defined constants.

C++ The remainder of this section pertains to C++ only.

A `const` object can appear in a constant expression if it is an integer and it is initialized to a constant. The following example demonstrates this.

```
const int k = 10;
int ary[k];     /* allowed in C++, not legal in C */
```

In C++, a `const` object can be defined in header files because a `const` object has internal linkage by default.

`const` Pointers

The keyword **const** for pointers can appear before the type, after the type, or in both places. The following are legal declarations:

```
const int * ptr1;       /* A pointer to a constant integer:
                             the value pointed to cannot be changed  */
int * const ptr2;       /* A constant pointer to integer:
                             the integer can be changed, but ptr2
                             cannot point to anything else          */
const int * const ptr3; /* A constant pointer to a constant integer:
                             neither the value pointed to
                             nor the pointer itself can be changed   */
```

Declaring an object to be `const` means that the `this` pointer is a pointer to a `const` object. A `const this` pointer can by used only with `const` member functions.

`const` Member Functions

Declaring a member function **const** means that the `this` pointer is a pointer to a `const` object. Data members of the class will be `const` within that function. The function is still able to change the value, but requires a `const_cast` to do so:

```
void foo::p() const{
   member = 1;                        // illegal
   const_cast <int&> (member) = 1;   // a bad practice but legal
}
```

A better technique would be to declare `member` **mutable**.

The volatile Type Qualifier

The **volatile** qualifier maintains consistency of memory access to data objects. Volatile objects are read from memory each time their value is needed, and written back to memory each time they are changed. The **volatile** qualifier declares a data object that can have its value changed in ways outside the control or detection of the compiler (such as a variable updated by the system clock). The compiler is thereby notified not to apply certain optimizations to code referring to the object.

Accessing any lvalue expression that is **volatile**-qualified produces a side effect. A side effect means that the state of the execution environment changes.

References to an object of type ″pointer to **volatile**″ may be optimized, but no optimization can occur to references to the object to which it points. An explicit cast must be used to assign a value of type ″pointer to **volatile**`T`″ to an object of type ″pointer to `T`″. The following shows valid uses of **volatile** objects.

```
volatile int * pvol;
int *ptr;
pvol = ptr;              /* Legal */
ptr = (int *)pvol;       /* Explicit cast required */
```

▶ C A signal-handling function may store a value in a variable of type `sig_atomic_t`, provided that the variable is declared **volatile**. This is an exception to the rule that a signal-handling function may not access variables with static storage duration.

The restrict Type Qualifier

▶ C The **restrict** type qualifier may only be applied to a pointer. A pointer declaration that uses this type qualifier establishes a special association between the pointer and the object it accesses, making that pointer and expressions based on that pointer, the only ways to directly or indirectly access the value of that object.

A pointer is the address of a location in memory. More than one pointer can access the same chunk of memory and modify it during the course of a program. The **restrict** type qualifier is an indication to the compiler that, if the memory addressed by the **restrict**-qualified pointer is modified, no other pointer will access that same memory. The compiler may choose to optimize code involving **restrict**-qualified pointers in a way that might otherwise result in incorrect behavior. It is the responsibility of the programmer to ensure that **restrict**-qualified pointers are used as they were intended to be used. Otherwise, undefined behavior may result.

If a particular chunk of memory is not modified, it can be aliased through more than one restricted pointer.

The following example shows restricted pointers as parameters of foo(), and how an unmodified object can be aliased through two restricted pointers.

```
void foo(int n, int * restrict a, int * restrict b, int * restrict c)
{
    int i;
    for (i = 0; i < n; i++)
        a[i] = b[i] + c[i];
}
```

Assignments between restricted pointers are limited, and no distinction is made between a function call and an equivalent nested block.

```
{
      int * restrict x;
      int * restrict y;
      x = y; // undefined
      {
         int * restrict x1 = x; // okay
         int * restrict y1 = y; // okay
         x = y1;  // undefined
      }
   }
```

In nested blocks containing restricted pointers, only assignments of restricted pointers from outer to inner blocks are allowed. The exception is when the block in which the restricted pointer is declared finishes execution. At that point in the program, the value of the restricted pointer can be carried out of the block in which it was declared.

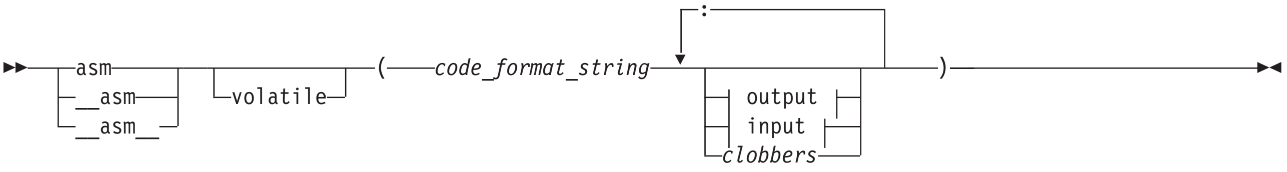
▶ Linux ▶ C++ C++ supports the **restrict** keyword as a non-orthogonal language extension for compatibility with C99. The compiler consumes and ignores the keyword, but issues a diagnostic message for incorrect usage. It is non-orthogonal because an existing C++ program can use *restrict* as a variable name.

The asm Statement

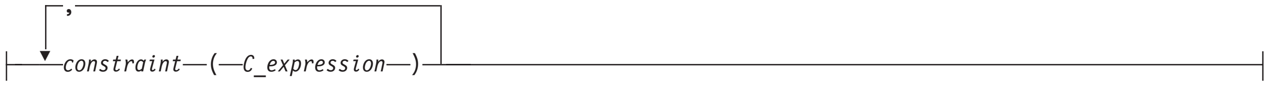
The keyword **asm** stands for assembly code. When compiled under strict language levels, the compiler recognizes and ignores the keyword **asm** in a declaration.

▶ Linux Under extended language levels, the compiler provides partial support for embedded assembly code fragments among C and C++ source statements. This extension has been implemented for use in general system programming code, in the kernel and device drivers, which were originally developed with GNU C.

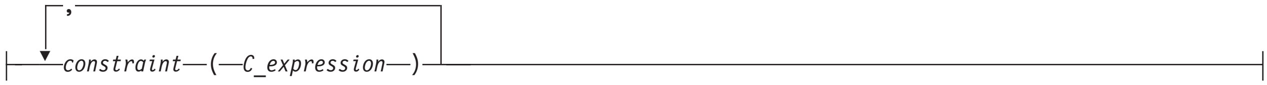
The syntax is as follows:



input:



output:



where

volatile
Instructs the compiler that the assembler instructions may update memory not listed in *output*, *input*, or *clobbers*.

code_format_string
Is the source text of the `asm` instructions and is a string literal similar to a `printf` format specifier.

input Is a comma-separated list of input operands.

output Is a comma-separated list of output operands.

clobbers
Is a comma-separated list of register names enclosed in double quotes. These are registers that can be updated by the `asm` instruction.

constraint
Is a string literal specifying the constraints for the operand, one character per constraint.

C_expression
Is a C or C++ expression whose value is used as the operand for the `asm` instruction. Output operands must be modifiable lvalues.

The following constraints are supported.

= Write-only operand.
+ Read and write operand.
& An operand may be modified before the instruction is finished using the input operands; a register that is used as input should not be reused here.
b Use a general register other than zero.
f Use a floating-point register.
g Use a general register, memory, or immediate operand.
i An immediate integer operand.
m A memory operand supported by the machine.
n Handle in the same way as *i*.
o Handle in the same way as *m*.
r Use a general register.
v Use a vector register.
0, 1, 2, ...66
A matching constraint. Allocate the same register in output as in the corresponding input.

I, J, K, M, N, O, P, G, S, T
Constant values. Fold the expression in the operand and substitute the value into the % specifier.

Restrictions

The assembler instructions must be self-contained within an **asm** statement. The **asm** statement can only be used to generate instructions. All connections to the rest of the program must be established through the *output* and *input* operand list. In particular:

- Branching to a label in another **asm** statement is not supported.
- Referencing an external symbol directly, without going through the operand list, is not supported.
- Pseudo-operators and directives, such as instructions with the suffix `.section`, `.text`, or `.data`, are not supported.
- The total number of instructions in one **asm** statement cannot exceed 63. The instruction count must also include the instructions generated by the compiler to handle the operands in the operand list.

Incomplete Types

The following are incomplete types:

- Type **void**
- Array of unknown size
- Arrays of elements that are of incomplete type
- Structure, union, or enumerations that have no definition
- C++ Pointers to class types that are declared but not defined
- C++ Classes that are declared but not defined

void is an incomplete type that cannot be completed. Incomplete structure or union and enumeration tags must be completed before being used to declare an object, although you can define a pointer to an incomplete structure or union.

C An array with an unspecified size is an incomplete type. However, if, instead of a constant expression, the array size is specified by [*], indicating a variable length array, the size is considered as having been specified, and the array type is then considered a complete type.

C If the function declarator is not part of a definition of that function, parameters may have incomplete type. The parameters may also have variable length array type, indicated by the [*] notation.

The following examples illustrate incomplete types:

```
void *incomplete_ptr;
struct dimension linear; /* no previous definition of dimension */
```

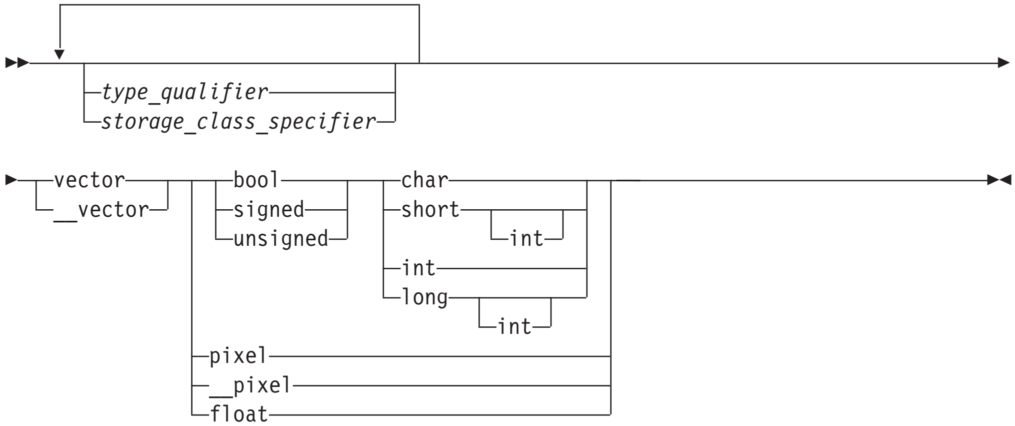
void is an incomplete type that cannot be completed. Incomplete structure, union, or enumeration tags must be completed before being used to declare an object. However, you can define a pointer to an incomplete structure or union.

AltiVec Types

Linux XL C/C++ supports the programming model of AltiVec technology through non-orthogonal language extensions. AltiVec data types are referred to as *vector types*. The IBM implementation of the AltiVec Programming Interface specification is an extended syntax that allows type qualifiers and storage class specifiers to precede the keyword **vector** (or its alternate spelling, **__vector**) in a declaration.

The keyword **vector** is recognized in a declaration context only when used as a type specifier and when the program is compiled with the option `-qaltivec`. The other AltiVec keywords, **pixel** and **bool**, are recognized as valid type specifiers only when preceded by the keyword **vector**. To keep your source code maximally portable, avoid using *vector*, *pixel*, or *bool* as keywords or identifiers in your program. Use the underscore versions of the specifiers **vector** and **pixel** (**__vector** and **__pixel**) in declarations.

Most of the legal forms of the syntax are captured in the following diagram. Some variations have been omitted from the diagram for the sake of clarity: type qualifiers such as **const** and storage class specifiers such as **static** can appear in any order within the declaration, as long as neither immediately follows the keyword **vector** (or **__vector**).



Notes:

1. The AltiVec specification has deprecated the **long** type specifier in a vector context.
2. Duplicate type specifiers are ignored in a vector declaration context. In particular, **long long** is treated as **long**.
3. A **long** vector type is compatible with the corresponding **int** vector type.

Restrictions

A `typedef` name as a type specifier is not allowed in a vector declaration context. However, a **vector** type can be used in a `typedef` declaration, and the resulting `typedef` name can be used in the usual way.

Alignment

All vector types are aligned on a 16-byte boundary. An aggregate that contains one or more vector types is aligned on a 16-byte boundary, and padded, if necessary, so that each member of vector type is also 16-byte aligned.

Pointers

The indirection operator * has been extended to handle pointer to vector types. A vector pointer should point to a memory location that has 16-byte alignment. However, the compiler does not enforce this constraint. Dereferencing a vector pointer maintains the vector type and its 16-byte alignment. If a program dereferences a vector pointer that does not contain a 16-byte aligned address, the behavior is undefined.

Pointer arithmetic is defined for pointer to vector types. Given:

```
vector unsigned int *v;
```

the expression `v + 1` represents a pointer to the vector following `v`.

Type Casting

Vector types can be cast to other vector types. The cast does not perform a conversion: it preserves the 128-bit pattern, but not necessarily the value. A cast between a vector type and a scalar type is not allowed.

Vector pointers and pointers to non-vector types can be cast back and forth to each other. When a pointer to a non-vector type is cast to a vector pointer, the address should be 16-byte aligned. The referenced object of the pointer to a non-vector type can be aligned on a sixteen-byte boundary by using either the `__align` specifier or `__attribute__((aligned(16)))`. Only vector types have a natural 16-byte alignment.

Initialization

A vector type is initialized by a vector literal or any expression having the same vector type. For example:

```
vector unsigned int v1;
vector unsigned int v2 = (vector unsigned int)(10);
v1 = v2;
```

▶ Linux A vector type can be initialized by an initializer list. The number of elements in a braced initializer list must be less than or equal to the number of elements of the vector type. Any uninitialized element will be initialized to zero.

Examples:

```
vector unsigned int v1 = {1};
  // initialize the first 4 bytes of v1 with 1 and the remaining 12 bytes with zeros
vector unsigned int v2 = {1,2};
  // initialize the first 8 bytes of v1 with 1 and 2 and the remaining 8 bytes with zeros
vector unsigned int v3 = {1,2,3,4};
  // equivalent to the vector literal (vector unsigned int) (1,2,3,4)
```

Unlike vector literals, the elements in the initializer list do not have to be constant expressions unless the initialized vector variable is static. Thus, the following is legal:

```
int i=1;
int foo() { return 2; }
int main()
```

```
{
   vector unsigned int v1 = {i, foo()};
   return 0;
}
```

Related References

- “Vector Literals” on page 29
- Appendix C, “AltiVec Data Types and Literals,” on page 397
- “The __align Specifier” on page 34

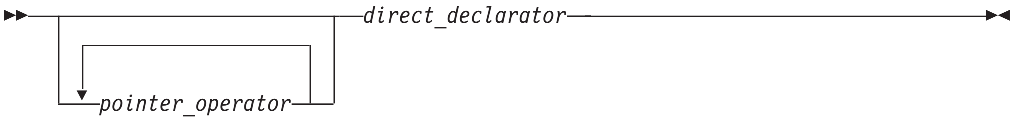
Chapter 4. Declarators

A *declarator* designates a data object or function. Declarators appear in most data definitions and declarations and in some type definitions.

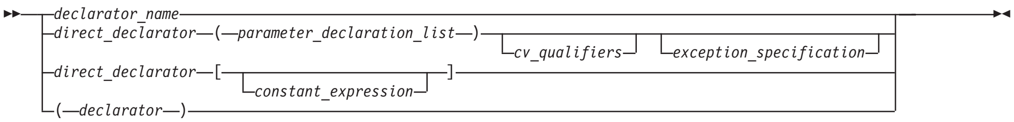
In a declarator, you can specify the type of an object to be an array, a pointer, or a reference. You can also perform initialization in a declarator.

A declarator has the form:

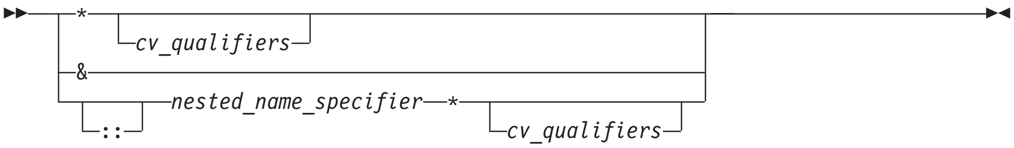
declarator



direct_declarator

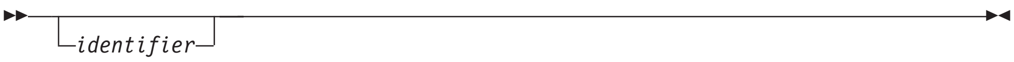


pointer_operator



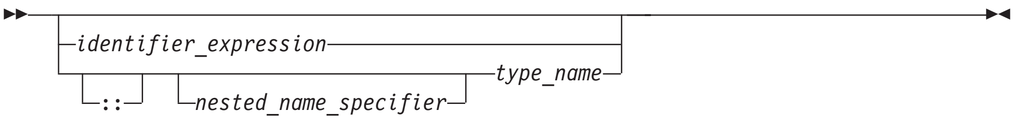
C The syntax for a declarator name in C:

declarator_id



C++ The syntax for a declarator name in C++:

declarator_id



Notes on the declarator syntax

- The *cv_qualifiers* variable represents one or a combination of **const** and **volatile**. In C, you cannot declare or define a **volatile** or **const** function. However, in C++, you can qualify a nonstatic member function with a cv-qualifier **const** or **volatile**.
- C++ The variables *exception_specification* and *nested_name_specifier* and the scope resolution operator `::` are available only in C++.
- C++ An *identifier_expression* can be a qualified or unqualified identifier. The complexity added by scope resolution operator, templates, and other advanced features does not exist for C.
- C++ A *nested_name_specifier* is a qualified identifier expression.

The following table provides some examples of declarators:

Example	Description
`int owner`	`owner` is an **int** data object.
`int *node`	`node` is a pointer to an **int** data object.
`int names[126]`	`names` is an array of 126 **int** elements.
`int *action( )`	`action` is a function returning a pointer to an **int**.
`volatile int min`	`min` is an **int** that has the **volatile** qualifier.
`int * volatile volume`	`volume` is a **volatile** pointer to an **int**.
`volatile int * next`	`next` is a pointer to a **volatile int**.
`volatile int * sequence[5]`	`sequence` is an array of five pointers to **volatile int** objects.
`extern const volatile int clock`	`clock` is a constant and volatile integer with static storage duration and external linkage.

Initializers

An *initializer* is an optional part of a data declaration that specifies an initial value of a data object. The initializers that are legal for a particular declaration depend on the type and storage class of the object to be initialized.

The initialization properties and special requirements of each data type are described in the section for that data type.

The initializer consists of the = symbol followed by an initial *expression* or a brace-enclosed list of initial expressions separated by commas. Individual expressions must be separated by commas, and groups of expressions can be enclosed in braces and separated by commas. Braces ({ }) are optional if the initializer for a character string is a string literal. The number of initializers must not be greater than the number of elements to be initialized. The initial expression evaluates to the first value of the data object.

To assign a value to an arithmetic or pointer type, use the simple initializer: = *expression*. For example, the following data definition uses the initializer `= 3` to set the initial value of `group` to 3:

```
int group = 3;
```

For unions, structures, and aggregate classes (classes with no constructors, base classes, virtual functions, or private or protected members), the set of initial expressions must be enclosed in braces unless the initializer is a string literal.

In an array, structure, or union initialized using a brace-enclosed initializer list, any members or subscripts that are not initialized are implicitly initialized to zero of the appropriate type.

Example

In the following example, only the first eight elements of the array `grid` are explicitly initialized. The remaining four elements that are not explicitly initialized are initialized as if they were explicitly initialized to zero.

```
static short grid[3] [4] = {0, 0, 0, 1, 0, 0, 1, 1};
```

The initial values of `grid` are:

Element	Value	Element	Value
grid[0] [0]	0	grid[1] [2]	1
grid[0] [1]	0	grid[1] [3]	1
grid[0] [2]	0	grid[2] [0]	0
grid[0] [3]	1	grid[2] [1]	0
grid[1] [0]	0	grid[2] [2]	0
grid[1] [1]	0	grid[2] [3]	0

C++ The remainder of this section pertains to C++ only.

In C++, you can initialize variables at namespace scope with nonconstant expressions. In C, you cannot do the same at global scope.

If your code jumps over declarations that contain initializations, the compiler generates an error. For example, the following code is not valid:

```
goto skiplabel;    // error - jumped over declaration
int i = 3;         //   and initialization of i

skiplabel: i = 4;
```

You can initialize classes in external, static, and automatic definitions. The initializer contains an = (equal sign) followed by a brace-enclosed, comma-separated list of values. You do not need to initialize all members of a class.

Pointers

A *pointer* type variable holds the address of a data object or a function. A pointer can refer to an object of any one data type; it cannot refer to a bit field or a reference. A pointer is classified as a scalar type, which means that it can hold only one value at a time.

Some common uses for pointers are:
- To access dynamic data structures such as linked lists, trees, and queues.
- To access elements of an array or members of a structure or C++ class.
- To access an array of characters as a string.
- To pass the address of a variable to a function. (In C++, you can also use a reference to do this.) By referencing a variable through its address, a function can change the contents of that variable.

C The remainder of this section pertains to C only.

You cannot use pointers to reference objects that are declared with the **register** storage class specifier.

Two pointer types with the same type qualifiers are compatible if they point to objects of compatible types. The composite type for two compatible pointer types is the similarly qualified pointer to the composite type.

Declaring Pointers

The following example declares pcoat as a pointer to an object having type **long**:

```
long *pcoat;
```

If the keyword **volatile** appears before the *, the declarator describes a pointer to a **volatile** object. If the keyword **volatile** appears between the * and the identifier, the declarator describes a **volatile** pointer. The keyword **const** operates in the same manner as the **volatile** keyword. In the following example, pvolt is a constant pointer to an object having type **short**:

```
extern short * const pvolt;
```

The following example declares pnut as a pointer to an **int** object having the **volatile** qualifier:

```
extern int volatile *pnut;
```

The following example defines psoup as a **volatile** pointer to an object having type **float**:

```
float * volatile psoup;
```

The following example defines pfowl as a pointer to an enumeration object of type bird:

```
enum bird *pfowl;
```

The next example declares pvish as a pointer to a function that takes no parameters and returns a **char** object:

```
char (*pvish)(void);
```

Assigning Pointers

When you use pointers in an assignment operation, you must ensure that the types of the pointers in the operation are compatible.

The following example shows compatible declarations for the assignment operation:

```
float subtotal;
float * sub_ptr;
/* ... */
sub_ptr = &subtotal;
printf("The subtotal is %f\n", *sub_ptr);
```

The next example shows incompatible declarations for the assignment operation:

```
double league;
int * minor;
/* ... */
minor = &league;     /* error */
```

Initializing Pointers

The initializer is an = (equal sign) followed by the expression that represents the address that the pointer is to contain. The following example defines the variables time and speed as having type **double** and amount as having type pointer to a **double**. The pointer amount is initialized to point to total:

```
double total, speed, *amount = &total;
```

The compiler converts an unsubscripted array name to a pointer to the first element in the array. You can assign the address of the first element of an array to a pointer by specifying the name of the array. The following two sets of definitions are equivalent. Both define the pointer student and initialize student to the address of the first element in section:

```
int section[80];
int *student = section;
```

is equivalent to:

```
int section[80];
int *student = &section[0];
```

You can assign the address of the first character in a string constant to a pointer by specifying the string constant in the initializer.

The following example defines the pointer variable string and the string constant "abcd". The pointer string is initialized to point to the character a in the string "abcd".

```
char *string = "abcd";
```

The following example defines weekdays as an array of pointers to string constants. Each element points to a different string. The pointer weekdays[2], for example, points to the string "Tuesday".

```
static char *weekdays[ ] =
      {
        "Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"
      };
```

A pointer can also be initialized to null using any integer constant expression that evaluates to 0, for example char * a=0;. Such a pointer is a *null pointer*. It does not point to any object.

Using Pointers

Two operators are commonly used in working with pointers, the address (&) operator and the indirection (*) operator. You can use the & operator to refer to the address of an object. For example, the assignment in the following function assigns the address of x to the variable p_to_int. The variable p_to_int has been defined as a pointer:

```
void f(int x, int *p_to_int)
{
  p_to_int = &x;
}
```

The * (indirection) operator lets you access the value of the object a pointer refers to. The assignment in the following example assigns to y the value of the object that p_to_float points to:

```
void g(float y, float *p_to_float) {
  y = *p_to_float;
}
```

The assignment in the following example assigns the value of z to the variable that *p_to_char references:

```
void h(char z, char *p_to_char) {
  *p_to_char = z;
}
```

Pointer Arithmetic

You can perform a limited number of arithmetic operations on pointers. These operations are:

- Increment and decrement
- Addition and subtraction
- Comparison
- Assignment

The increment (++) operator increases the value of a pointer by the size of the data object the pointer refers to. For example, if the pointer refers to the second element in an array, the ++ makes the pointer refer to the third element in the array.

The decrement (--) operator decreases the value of a pointer by the size of the data object the pointer refers to. For example, if the pointer refers to the second element in an array, the -- makes the pointer refer to the first element in the array.

You can add an integer to a pointer but you cannot add a pointer to a pointer.

If the pointer p points to the first element in an array, the following expression causes the pointer to point to the third element in the same array:

```
p = p + 2;
```

If you have two pointers that point to the same array, you can subtract one pointer from the other. This operation yields the number of elements in the array that separate the two addresses that the pointers refer to.

You can compare two pointers with the following operators: ==, !=, <, >, <=, and >=.

Pointer comparisons are defined only when the pointers point to elements of the same array. Pointer comparisons using the == and != operators can be performed even when the pointers point to elements of different arrays.

You can assign to a pointer the address of a data object, the value of another compatible pointer or the NULL pointer.

Example Program Using Pointers

The following program contains pointer arrays:

```
/*********************************************************************
**   Program to search for the first occurrence of a specified      **
**   character string in an array of character strings.             **
*********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#define  SIZE  20

int main(void)
{
   static char *names[ ] = { "Jim", "Amy", "Mark", "Sue", NULL };
   char * find_name(char **, char *);
   char new_name[SIZE], *name_pointer;

   printf("Enter name to be searched.\n");
   scanf("%s", new_name);
   name_pointer = find_name(names, new_name);
   printf("name %s%sfound\n", new_name,
          (name_pointer == NULL) ? " not " : " ");
} /* End of main */



/********************************************************************
**     Function find_name.  This function searches an array of     **
**     names to see if a given name already exists in the array.   **
**     It returns a pointer to the name or NULL if the name is     **
**     not found.                                                  **
**                                                                 **
** char **arry is a pointer to arrays of pointers (existing names) **
** char *strng is a pointer to character array entered (new name)  **
********************************************************************/

char * find_name(char **arry, char *strng)
{
   for (; *arry != NULL; arry++)        /* for each name          */
   {
      if (strcmp(*arry, strng) == 0)     /* if strings match       */
         return(*arry);                  /* found it!              */
   }
   return(*arry);                        /* return the pointer     */
} /* End of find_name */
```

Interaction with this program could produce the following sessions:

Output	`Enter name to be searched.`
Input	`Mark`
Output	`name Mark found`

or:

Output	`Enter name to be searched.`
Input	`Deborah`
Output	`name Deborah not found`

Arrays

An *array* is a collection of objects of the same data type. Individual objects in an array, called *elements,* are accessed by their position in the array. The subscripting operator ([]) provides the mechanics for creating an index to array elements. This form of access is called *indexing* or *subscripting*. An array facilitates the coding of repetitive tasks by allowing the statements executed on each element to be put into a loop that iterates through each element in the array.

The C and C++ languages provide limited built-in support for an array type: reading and writing individual elements. Assignment of one array to another, the comparison of two arrays for equality, returning self-knowledge of size are operations unsupported by either language.

An array type describes contiguously allocated memory for a set of objects of a particular type. The array type is derived from the type of its elements, in what is called *array type derivation*. If array objects are of incomplete type, the array type is also considered incomplete.

Array elements may not be of type **void** or of function type. However, arrays of pointers to functions are allowed. In C++, array elements may not be of reference type or of an abstract class type.

▶ C Two array types that are similarly qualified are compatible if the types of their elements are compatible. For example,

```
char ex1[25];
const char ex2[25];
```

are not compatible. The composite type of two compatible array types is an array with the composite element type. The sizes of both original types must be equivalent if they are known. If the size of only one of the original array types is known, then the composite type has that size. For example, suppose:

```
char ex3[];
char ex4[42];
```

The composite type of `ex3` and `ex4` is `char[42]`. If one of the original types is a variable length array, the composite type is that type.

Except in certain contexts, an unsubscripted array name (for example, `region` instead of `region[4]`) represents a pointer whose value is the address of the first element of the array, provided that the array has previously been declared. The exceptions are when the array name passes the array itself. For example, the array name passes the entire array when it is the operand of the **sizeof** operator or the address (**&**) operator.

Similarly, an array type in the parameter list of a function is converted to the corresponding pointer type. Information about the size of the argument array is lost when the array is accessed from within the function body.

▶ C To preserve this information, which is useful for optimization, you may declare the index of the argument array using the **static** keyword. The constant expression specifies the minimum pointer size that can be used as an assumption for optimizations.

This particular usage of the **static** keyword is highly prescribed. The keyword may only appear in the outermost array type derivation and only in function parameter declarations. If the caller of the function does not abide by these restrictions, the behavior is undefined.

This language feature is available at the C99 language level.

The following examples show how the feature might be used.

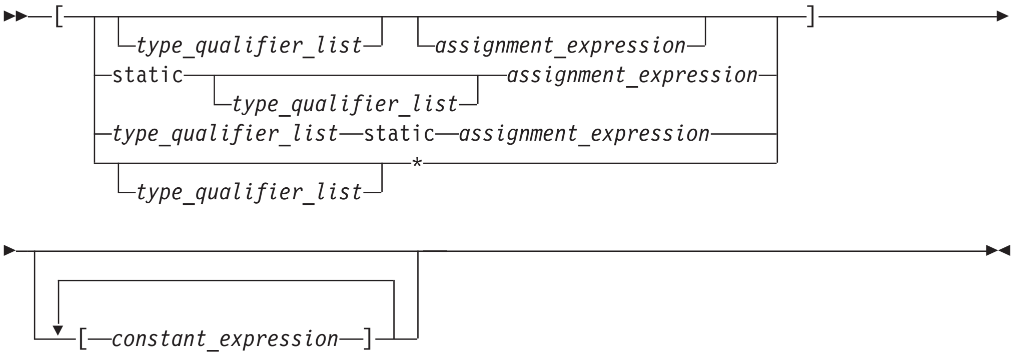
```
void foo(int arr [static 10]);       /* arr points to the first of at least
                                           10 ints                            */
void foo(int arr [const 10]);        /* arr is a const pointer                */
void foo(int arr [static const i]);  /* arr points to at least i ints;
```

```
                                           i is computed at run time.       */
void foo(int arr [const static i]);  /* alternate syntax to previous example */
void foo(int arr [const]);           /* const pointer to int                 */
```

Declaring Arrays

The array declarator contains an identifier followed by an optional *subscript declarator*. An identifier preceded by an asterisk (*) is an array of pointers.

A subscript declarator has the form:



where *constant_expression is a constant integer expression, indicating the size of the array, which must be positive.*

C If the declaration appears in block or function scope, a nonconstant expression can be specified for the array subscript declarator, and the array is considered a variably modified type. An asterisk within the brackets of the array subscripting operator indicates a variable length array of unspecified size. In this case, the array is considered a variably modified type that can only be used in functions declarations that are not definitions (that is, in declarations with function prototype scope).

The subscript declarator describes the number of dimensions in the array and the number of elements in each dimension. Each bracketed expression, or subscript, describes a different dimension and must be a constant expression.

The following example defines a one-dimensional array that contains four elements having type **char**:

```
char
list[4];
```

The first subscript of each dimension is `0`. The array `list` contains the elements:

```
list[0]
list[1]
list[2]
list[3]
```

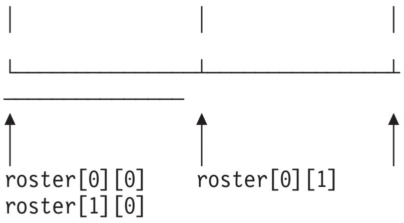
The following example defines a two-dimensional array that contains six elements of type **int**:

```
int
roster[3][2];
```

Multidimensional arrays are stored in row-major order. When elements are referred to in order of increasing storage location, the last subscript varies the fastest. For example, the elements of array `roster` are stored in the order:

```
roster[0][0]
roster[0][1]
roster[1][0]
roster[1][1]
roster[2][0]
roster[2][1]
```

In storage, the elements of `roster` would be stored as:



You can leave the first (and only the first) set of subscript brackets empty in

- Array definitions that contain initializations
- **extern** declarations
- Parameter declarations

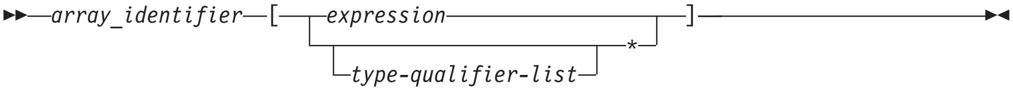
In array definitions that leave the first set of subscript brackets empty, the initializer determines the number of elements in the first dimension. In a one-dimensional array, the number of initialized elements becomes the total number of elements. In a multidimensional array, the initializer is compared to the subscript declarator to determine the number of elements in the first dimension.

Variable Length Arrays

A variable length array is an array of automatic storage duration whose length is determined at run time. The variable length array type provides a construct for allocating the right amount of storage, which can only be determined when the application is actually run.

C++ C++ extends Standard C++ to support variable length arrays for compatibility with C99. This extension does not include support for references to a variable length array type; neither may a function parameter be a reference to a variable length array type.

A variable length array can be written as:



If the size of the array is indicated by `*` instead of an expression, the variable length array is considered to be of unspecified size. Such arrays are considered complete types, but can only be used in declarations of function prototype scope.

A variable length array and a pointer to a variable length array are considered *variably modified types*. Declarations of variably modified types must be at either block scope or function prototype scope. Array objects declared with the `extern` storage class specifier cannot be of variable length array type. Array objects declared with the `static` storage class specifier can be a pointer to a variable length array, but not an actual variable length array. The identifiers declared with a variably modified type must be ordinary identifiers and therefore cannot be members of structures or unions. A variable length array cannot be initialized.

A variable length array can be the operand of a `sizeof` expression. In this case, the operand is evaluated at run time, and the size is neither an integer constant nor a constant expression, even though the size of each instance of a variable array does not change during its lifetime.

A variable length array can be used in a `typedef` expression. The `typedef` name will have only block scope. The length of the array is fixed when the `typedef` name is defined, not each time it is used.

A function parameter can be a variable length array. The necessary size expressions must be provided in the function definition. The compiler evaluates the size expression of a variably modified parameter on entry to the function. For a function declared with a variable length array as a parameter, as in the following,

```
void f(int x, int a[][x]);
```

the size of the variable length array argument must match that of the function definition.

Related References
- “Calling Functions and Passing Arguments” on page 177

Initializing Arrays

The initializer for an array is a comma-separated list of constant expressions enclosed in braces (`{ }`). The initializer is preceded by an equal sign (`=`). You do not need to initialize all elements in an array. If an array is partially initialized, elements that are not initialized receive the value `0` of the appropriate type. The same applies to elements of arrays with static storage duration. (All file-scope variables and function-scope variables declared with the **static** keyword have static storage duration.)

The following definition shows a completely initialized one-dimensional array:

```
static int number[3] = { 5, 7, 2 };
```

The array `number` contains the following values: `number[0]` is `5`, `number[1]` is `7`; `number[2]` is `2`. When you have an expression in the subscript declarator defining the number of elements (in this case `3`), you cannot have more initializers than the number of elements in the array.

The following definition shows a partially initialized one-dimensional array:

```
static int number1[3] = { 5, 7 };
```

The values of `number1` are:`number1[0]` and `number1[1]` are the same as in the previous definition, but `number1[2]` is `0`.

Instead of an expression in the subscript declarator defining the number of elements, the following one-dimensional array definition defines one element for each initializer specified:

```
static int item[ ] = { 1, 2, 3, 4, 5 };
```

The compiler gives `item` the five initialized elements, because no size was specified and there are five initializers.

You can initialize a one-dimensional character array by specifying:

- A brace-enclosed comma-separated list of constants, each of which can be contained in a character
- A string constant (Braces surrounding the constant are optional)

Initializing a string constant places the null character (`\0`) at the end of the string if there is room or if the array dimensions are not specified.

The following definitions show character array initializations:

```
static char name1[ ] = { 'J', 'a', 'n' };
static char name2[ ] = { "Jan" };
static char name3[4] = "Jan";
```

These definitions create the following elements:

Element	Value	Element	Value	Element	Value
name1[0]	J	name2[0]	J	name3[0]	J
name1[1]	a	name2[1]	a	name3[1]	a
name1[2]	n	name2[2]	n	name3[2]	n
		name2[3]	\0	name3[3]	\0

Note that the following definition would result in the null character being lost:

```
static char name3[3]="Jan";
```

▶ C++ When initializing an array of characters with a string, the number of characters in the string — including the terminating `'\0'` — must not exceed the number of elements in the array.

You can initialize a multidimensional array using any of the following techniques:

- Listing the values of all elements you want to initialize, in the order that the compiler assigns the values. The compiler assigns values by increasing the subscript of the last dimension fastest. This form of a multidimensional array initialization looks like a one-dimensional array initialization. The following definition completely initializes the array `month_days`:

  ```
  static month_days[2][12] =
  {
   31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31,
   31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
  };
  ```

- Using braces to group the values of the elements you want initialized. You can put braces around each element, or around any nesting level of elements. The following definition contains two elements in the first dimension (you can consider these elements as rows). The initialization contains braces around each of these two elements:

  ```
  static int month_days[2][12] =
  {
   { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
   { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
  };
  ```

- Using nested braces to initialize dimensions and elements in a dimension selectively.

The following definition explicitly initializes six elements in a 12-element array:

```
static int matrix[3][4] =
   {
     {1, 2},
     {3, 4},
     {5, 6}
   };
```

The initial values of `matrix` are shown in the following table. All other elements are initialized to zero.

Element	Value	Element	Value
matrix[0][0]	1	matrix[1][2]	0
matrix[0][1]	2	matrix[1][3]	0
matrix[0][2]	0	matrix[2][0]	5
matrix[0][3]	0	matrix[2][1]	6
matrix[1][0]	3	matrix[2][2]	0
matrix[1][1]	4	matrix[2][3]	0

Initializing Arrays Using Designated Initializers

▶ C C supports designated initializers for aggregate types. A *designator* points out a particular array element to be initialized, and is of the form "[*index*]", where *index* is a constant expression. A *designator list* is a combination of one or more designators for any of the aggregate types. A designator list followed by an equal sign constitutes a *designation*.

In the absence of designations, initialization of an array occurs in the order indicated by the initializer. When a designation appears in an initializer, the array element indicated by the designator is initialized, and subsequent initializations proceed forward in initializer-list order, overriding any previously initialized array element, and initializing to zero any array elements that are not explicitly initialized.

The declaration syntax without a designated initializer uses braces to indicate initializer lists, but is referred to as a *bracketed form*. The fully bracketed and minimally bracketed forms of initialization are less likely to be misunderstood. The following are valid declarations of the multidimensional array `matrix` that achieve the same thing. All array elements that are not explicitly initialized, such as the entire row beginning with `matrix[3][0][0]`, are initialized to zero.

```
/* minimally bracketed form */
int matrix[4][3][2] = {
    1, 0, 0, 0, 0, 0,
    2, 3, 0, 0, 0, 0,
    4, 5, 6
};

/* fully bracketed form */
int matrix[4] [3] [2] = {
    {
        { 1 },
    },
    {
        { 2, 3 },
    },
    {
        { 4, 5 },
        { 6 }
    }
};
```

```
/* incompletely but consistently bracketed initialization */
int matrix[4] [3] [2] = {
    { 1 },
    { 2, 3 },
    { 4, 5, 6 }
};
```

The overriding of previous subobject initializations during an array initialization is necessary behavior for the designated initializer. To illustrate this, a single designator is used to ″allocate″ space from both ends of an array. The designated initializer, [MAX-5] = 8, means that the array element at subscript MAX-5 should be initialized to the value 8. The array subscripting brackets must enclose a constant expression.

```
int a[MAX] = {
      1, 3, 5, 7, 9, [MAX-5] = 8, 6, 4, 2, 0
};
```

If MAX is 15, a[5] through a[9] will be initialized to zero. If MAX is 7, a[2] through a[4] will first have the values 5, 7, and 9, respectively, which are overridden by the values 8, 6, and 4. In other words, if MAX is 7, the initialization would be the same as if the declaration had been written:

```
int a[MAX] = {
      1, 3, 8, 6, 4, 2, 0
};
```

Example Programs Using Arrays

The following program defines a floating-point array called prices.

The first for statement prints the values of the elements of prices. The second for statement adds five percent to the value of each element of prices, and assigns the result to total, and prints the value of total.

```
/**
 ** Example of one-dimensional arrays
 **/

#include <stdio.h>
#define  ARR_SIZE  5

int main(void)
{
  static float const prices[ARR_SIZE] = { 1.41, 1.50, 3.75, 5.00, .86 };
  auto float total;
  int i;

  for (i = 0; i < ARR_SIZE; i++)
  {
    printf("price = $%.2f\n", prices[i]);
  }

  printf("\n");

  for (i = 0; i < ARR_SIZE; i++)
  {
    total = prices[i] * 1.05;
    printf("total = $%.2f\n", total);
  }

  return(0);
}
```

This program produces the following output:

```
price = $1.41
price = $1.50
price = $3.75
price = $5.00
price = $0.86

total = $1.48
total = $1.57
total = $3.94
total = $5.25
total = $0.90
```

The following program defines the multidimensional array `salary_tbl`. A **for** loop prints the values of `salary_tbl`.

```
/**
 ** Example of a multidimensional array
 **/

#include <stdio.h>
#define  ROW_SIZE     3
#define  COLUMN_SIZE  5

int main(void)
{
  static int
  salary_tbl[ROW_SIZE][COLUMN_SIZE] =
  {
    {  500,  550,  600,  650,  700   },
    {  600,  670,  740,  810,  880   },
    {  740,  840,  940, 1040, 1140   }
  };
  int grade , step;

  for (grade = 0; grade < ROW_SIZE; ++grade)
   for (step = 0; step < COLUMN_SIZE; ++step)
   {
     printf("salary_tbl[%d] [%d] = %d\n",
            grade, step, salary_tbl[grade] [step]);
   }

   return(0);
}
```

This program produces the following output:

```
salary_tbl[0] [0] = 500
salary_tbl[0] [1] = 550
salary_tbl[0] [2] = 600
salary_tbl[0] [3] = 650
salary_tbl[0] [4] = 700
salary_tbl[1] [0] = 600
salary_tbl[1] [1] = 670
salary_tbl[1] [2] = 740
salary_tbl[1] [3] = 810
salary_tbl[1] [4] = 880
salary_tbl[2] [0] = 740
salary_tbl[2] [1] = 840
salary_tbl[2] [2] = 940
salary_tbl[2] [3] = 1040
salary_tbl[2] [4] = 1140
```

Function Specifiers

The function specifier **inline** is used to make a suggestion to the compiler to incorporate the code of a function into the code at the point of the call. Instead of creating a single set of the function instructions in memory, the compiler is supposed to copy the code from the inline function directly into the calling function. However, a standards-compliant compiler may ignore this suggestion for better optimization.

C++ The remainder of this section pertains to C++ only.

Both regular functions and member functions can be declared inline. A member function can be made inline by using the keyword **inline**, even if the function is declared outside of the class declaration.

The keywords **virtual** and **explicit** are used only in C++ function declarations as function specifiers.

The function specifier **virtual** can only be used in nonstatic member function declarations.

The function specifier **explicit** can only be used in declarations of constructors within a class declaration. It is used to control unwanted implicit type conversions when an object is being initialized. An explicit constructor differs from a non-explicit constructor in that an explicit constructor can only construct objects where direct initialization syntax or explicit casts are used.

References

C++ A *reference* is an alias or an alternative name for an object. All operations applied to a reference act on the object to which the reference refers. The address of a reference is the address of the aliased object.

A reference type is defined by placing the reference declarator `&` after the type specifier. You must initialize all references except function parameters when they are defined. Once defined, a reference cannot be reassigned. What happens when you try to reassign a reference turns out to be the assignment of a new value to the target.

Because arguments of a function are passed by value, a function call does not modify the actual values of the arguments. If a function needs to modify the actual value of an argument or needs to return more than one value, the argument must be *passed by reference* (as opposed to being *passed by value*). Passing arguments by reference can be done using either references or pointers. Unlike C, C++ does not force you to use pointers if you want to pass arguments by reference. The syntax of using a reference is somewhat simpler than that of using a pointer. Passing an object by reference enables the function to change the object being referred to without creating a copy of the object within the scope of the function. Only the address of the actual original object is put on the stack, not the entire object.

For example:

```
int f(int&);
int main()
{
      extern int i;
      f(i);
}
```

You cannot tell from the function call `f(i)` that the argument is being passed by reference.

References to `NULL` are not allowed.

Initializing References

The object that you use to initialize a reference must be of the same type as the reference, or it must be of a type that is convertible to the reference type. If you initialize a reference to a constant using an object that requires conversion, a temporary object is created. In the following example, a temporary object of type **float** is created:

```
int i;
const float& f = i; // reference to a constant float
```

When you initialize a reference with an object, you *bind* that reference to that object.

Attempting to initialize a nonconstant reference with an object that requires a conversion is an error.

Once a reference has been initialized, it cannot be modified to refer to another object. For example:

```
int num1 = 10;
int num2 = 20;

int &RefOne = num1;          // valid
int &RefOne = num2;          // error, two definitions of RefOne
RefOne = num2;                       // assign num2 to num1
int &RefTwo;                 // error, uninitialized reference
int &RefTwo = num2;          // valid
```

Note that the initialization of a reference is not the same as an assignment to a reference. Initialization operates on the actual reference by initializing the reference with the object it is an alias for. Assignment operates through the reference on the object referred to.

A reference can be declared without an initializer:
- When it is used in an parameter declaration
- In the declaration of a return type for a function call
- In the declaration of class member within its class declaration
- When the **extern** specifier is explicitly used

You cannot have references to any of the following:
- Other references
- Bit fields
- Arrays of references
- Pointers to references

Direct Binding

Suppose a reference r of type T is initialized by an expression e of type U.

The reference r is *bound directly* to e if the following statements are true:
- Expression e is an lvalue
- T is the same type as U, or T is a base class of U
- T has the same, or more, **const** or **volatile** qualifiers than U

The reference r is also bound directly to e if e can be implicitly converted to a type such that the previous list of statements is true.

Chapter 5. Expressions and Operators

Expressions are sequences of operators, operands, and punctuators that specify a computation. The evaluation of expressions is based on the operators that the expressions contain and the context in which they are used. An expression can result in a value and can produce side effects. A *side effect* is a change in the state of the execution environment.

Both ISO C and ISO C++ heed points in the execution sequence at which all side effects of previous evaluations are complete and no side effects of subsequent evaluations will have occurred. Such times are called *sequence points*. A scalar object may be modified only once between successive sequence points; otherwise, the result is undefined. Sequence points occur at the completion of all expressions that are not part of a larger expression, such as in the following situations:

- After the evaluation of the first operand of a logical AND &&, logical OR `||`, conditional ?:, or comma expression
- After the evaluation of the arguments in a function call
- At the end of a full declarator
- At the end of a full expression
- Before a library function returns
- After the actions of a formatted I/O function conversion specifier
- Before and after a call to a comparison function, and between any call to the comparison function and any movement of the objects passed as arguments to that function call

The term *full expression* can mean an initializer, an expression statement, the expression in a `return` statement, and the control expressions in a conditional, iterative, or `switch` statement. This includes each expression in a `for` statement.

▶ C++ C++ operators can be defined to behave differently when applied to operands of class type. This is called operator *overloading*. This chapter describes the behavior of operators that are not overloaded.

Related References
- “Lvalues and Rvalues” on page 103
- “Overloading Operators” on page 239

Operator Precedence and Associativity

Two operator characteristics determine how operands group with operators: *precedence* and *associativity*. Precedence is the priority for grouping different types of operators with their operands. Associativity is the left-to-right or right-to-left order for grouping operands to operators that have the same precedence. An operator’s precedence is meaningful only if other operators with higher or lower precedence are present. Expressions with higher-precedence operators are evaluated first. The grouping of operands can be forced by using parentheses.

For example, in the following statements, the value of `5` is assigned to both `a` and `b` because of the right-to-left associativity of the `=` operator. The value of `c` is assigned to `b` first, and then the value of `b` is assigned to `a`.

```
b = 9;
c = 5;
a = b = c;
```

Because the order of subexpression evaluation is not specified, you can explicitly force the grouping of operands with operators by using parentheses.

In the expression

```
a + b * c / d
```

the `*` and `/` operations are performed before `+` because of precedence. `b` is multiplied by `c` before it is divided by `d` because of associativity.

The following table lists the C and C++ language operators in order of precedence and shows the direction of associativity for each operator.

The C++ scope resolution operator (`::`) has the highest precedence. The comma operator has the lowest precedence. Operators that have the same rank have the same precedence.

Precedence and associativity of C and C++ operators

Rank	Right Associative?	Operator Function	Usage
1	yes	C++ global scope resolution	**::** *name_or_qualified name*
1		C++ class or namespace scope resolution	*class_or_namespace* **::** *member*
2		member selection	*object* **.** *member*
2		member selection	*pointer* **->** *member*
2		subscripting	*pointer* **[** *expr* **]**
2		function call	*expr* **(** *expr_list* **)**
2		value construction	*type* **(** *expr_list* **)**
2		postfix increment	*lvalue* **++**
2		postfix decrement	*lvalue* **--**
2	yes	C++ type identification	**typeid (** *type* **)**
2	yes	C++ type identification at run time	**typeid (** *expr* **)**
2	yes	C++ conversion checked at compile time	**static_cast <** *type* **> (** *expr* **)**
2	yes	C++ conversion checked at run time	**dynamic_cast <** *type* **> (** *expr* **)**
2	yes	C++ unchecked conversion	**reinterpret_cast <** *type* **> (** *expr* **)**
2	yes	C++ **const** conversion	**const_cast <** *type* **> (** *expr* **)**
3	yes	size of object in bytes	**sizeof** *expr*
3	yes	size of type in bytes	**sizeof (** *type* **)**
3	yes	prefix increment	**++** *lvalue*
3	yes	prefix decrement	**--** *lvalue*
3	yes	bitwise negation	**~** *expr*
3	yes	not	**!** *expr*
3	yes	unary minus	**-** *expr*
3	yes	unary plus	**+** *expr*
3	yes	address of	**&** *lvalue*
3	yes	indirection or dereference	* *expr*

Precedence and associativity of C and C++ operators

Rank	Right Associative?	Operator Function	Usage
3	yes	C++ create (allocate memory)	**new** *type*
3	yes	C++ create (allocate and initialize memory)	**new** *type* **(** *expr_list* **)** *type*
3	yes	C++ create (placement)	**new** *type* **(** *expr_list* **)** *type* (*expr_list*)
3	yes	C++ destroy (deallocate memory)	**delete** *pointer*
3	yes	C++ destroy array	**delete [ ]** *pointer*
3	yes	type conversion (cast)	**(** *type* **)** *expr*
4		member selection	*object* **.*** *ptr_to_member*
4		member selection	*object* **->*** *ptr_to_member*
5		multiplication	*expr* * *expr*
5		division	*expr* **/** *expr*
5		modulo (remainder)	*expr* **%** *expr*
6		binary addition	*expr* **+** *expr*
6		binary subtraction	*expr* **-** *expr*
7		bitwise shift left	*expr* **<<** *expr*
7		bitwise shift right	*expr* **>>** *expr*
8		less than	*expr* **<** *expr*
8		less than or equal to	*expr* **<=** *expr*
8		greater than	*expr* **>** *expr*
8		greater than or equal to	*expr* **>=** *expr*
9		equal	*expr* **==** *expr*
9		not equal	*expr* **!=** *expr*
10		bitwise AND	*expr* **&** *expr*
11		bitwise exclusive OR	*expr* **^** *expr*
12		bitwise inclusive OR	*expr* **|** *expr*
13		logical AND	*expr* **&&** *expr*
14		logical inclusive OR	*expr* **||** *expr*
15		conditional expression	*expr* **?** *expr* **:** *expr*
16	yes	simple assignment	*lvalue* **=** *expr*
16	yes	multiply and assign	*lvalue* ***=** *expr*
16	yes	divide and assign	*lvalue* **/=** *expr*
16	yes	modulo and assign	*lvalue* **%=** *expr*
16	yes	add and assign	*lvalue* **+=** *expr*
16	yes	subtract and assign	*lvalue* **-=** *expr*
16	yes	shift left and assign	*lvalue* **<<=** *expr*
16	yes	shift right and assign	*lvalue* **>>=** *expr*
16	yes	bitwise AND and assign	*lvalue* **&=** *expr*
16	yes	bitwise exclusive OR and assign	*lvalue* **^=** *expr*
16	yes	bitwise inclusive OR and assign	*lvalue* **|=** *expr*
17	yes	C++ throw expression	**throw** *expr*
18		comma (sequencing)	*expr* **,** *expr*

The order of evaluation for function call arguments or for the operands of binary operators is not specified. Avoid writing ambiguous expressions such as:

```
z = (x * ++y) / func1(y);
func2(++i, x[i]);
```

In the example above, ++y and func1(y) might not be evaluated in the same order by all C language implementations. If y has the value of 1 before the first statement, it is not known whether or not the value of 1 or 2 is passed to func1(). In the second statement, if i has the value of 1 before the expression is evaluated, it is not known whether x[1] or x[2] is passed as the second argument to func2().

Examples of Expressions and Precedence

The parentheses in the following expressions explicitly show how the compiler groups operands and operators.

```
total = (4 + (5 * 3));
total = (((8 * 5) / 10) / 3);
total = (10 + (5/3));
```

If parentheses did not appear in these expressions, the operands and operators would be grouped in the same manner as indicated by the parentheses. For example, the following expressions produce the same output.

```
total = (4+(5*3));
total = 4+5*3;
```

Because the order of grouping operands with operators that are both associative and commutative is not specified, the compiler can group the operands and operators in the expression:

```
total = price + prov_tax +
city_tax;
```

in the following ways (as indicated by parentheses):

```
total = (price + (prov_tax + city_tax));
total = ((price + prov_tax) + city_tax);
total = ((price + city_tax) + prov_tax);
```

The grouping of operands and operators does not affect the result unless one ordering causes an overflow and another does not. For example, if price = 32767, prov_tax = -42, and city_tax = 32767, and all three of these variables have been declared as integers, the third statement total = ((price + city_tax) + prov_tax) will cause an integer overflow and the rest will not.

Because intermediate values are rounded, different groupings of floating-point operators may give different results.

In certain expressions, the grouping of operands and operators can affect the result. For example, in the following expression, each function call might be modifying the same global variables.

```
a = b() + c() + d();
```

This expression can give different results depending on the order in which the functions are called.

If the expression contains operators that are both associative and commutative and the order of grouping operands with operators can affect the result of the expression, separate the expression into several expressions. For example, the following expressions could replace the previous expression if the called functions do not produce any side effects that affect the variable a.

```
a = b();
a += c();
a += d();
```

Lvalues and Rvalues

An *object* is a region of storage that can be examined and stored into. An *lvalue* is an expression that refers to such an object. An lvalue does not necessarily permit modification of the object it designates. For example, a **const** object is an lvalue that cannot be modified. The term *modifiable lvalue* is used to emphasize that the lvalue allows the designated object to be changed as well as examined. The following object types are lvalues, but not modifiable lvalues:

- An array type
- An incomplete type
- A **const**-qualified type
- An object is a structure or union type and one of its members has a **const**-qualified type

Because these lvalues are not modifiable, they cannot appear on the left side of an assignment statement.

The term *rvalue* refers to a data value that is stored at some address in memory. An *rvalue* is an expression that cannot have a value assigned to it. Both a literal constant and a variable can serve as an rvalue. When an lvalue appears in a context that requires an rvalue, the lvalue is implicitly converted to an rvalue. The reverse, however, is not true: an rvalue cannot be converted to an lvalue. Rvalues always have complete types or the void type.

C ISO C defines a *function designator* as an expression that has function type A function designator is distinct from an object type or an lvalue. It can be the name of a function or the result of dereferencing a function pointer. The C language also differentiates between its treatment of a function pointer and an object pointer.

C++ On the other hand, in C++, a function call that returns a reference is an lvalue. Otherwise, a function call is an rvalue expression. In C++, every expression produces an *lvalue*, an *rvalue*, or no value.

In both C and C++, certain operators require lvalues for some of their operands. The table below lists these operators and additional constraints on their usage.

Operator	Requirement
& (unary)	Operand must be an lvalue.
++ --	Operand must be an lvalue. This applies to both prefix and postfix forms.
= += -= *= %= <<= >>= &= ^= |=	Left operand must be an lvalue.

For example, all assignment operators evaluate their right operand and assign that value to their left operand. The left operand must be a modifiable lvalue or a reference to a modifiable object.

The address operator (&) requires an lvalue as an operand while the increment (++) and the decrement (--) operators require a modifiable lvalue as an operand. The following example shows expressions and their corresponding lvalues.

Expression	Lvalue
`x = 42`	`x`

Expression	Lvalue
`*ptr = newvalue`	`*ptr`
`a++`	`a`
▶ C++ `int& f()`	The function call to `f()`

▶ C The remainder of this section is platform-specific and pertains to C only.

When compiled with the GNU C language extensions enabled, compound expressions, conditional expressions, and casts are allowed as lvalues, provided that their operands are lvalues. The use of this language extension is deprecated for C++ code.

A compound expression can be assigned if the last expression in the sequence is an lvalue. The following expressions are equivalent:

```
(x + 1, y) *= 42;
x + 1, (y *=42);
```

The address operator can be applied to a compound expression, provided the last expression in the sequence is an lvalue. The following expressions are equivalent:

```
&(x + 1, y);
x + 1, &y;
```

A conditional expression can be a valid lvalue if its type is not void and both of its branches for true and false are valid lvalues. Casts are valid lvalues if the operand is an lvalue. The primary restriction is that you cannot take the address of an lvalue cast.

Related References
- "Lvalue-to-Rvalue Conversions" on page 150

Primary Expressions

Primary expressions fall into the following general categories:
- Names (identifiers)
- Literals (constants)
- Parenthesized expressions
- ▶ C++ The **this** pointer
- ▶ C++ Names qualified by the scope resolution operator (`::`)

Names

The value of a name depends on its type, which is determined by how that name is declared. The following table shows whether a name is an lvalue expression.

Primary expressions: Names

Name declared as	Evaluates to	Is an lvalue
Variable of arithmetic, pointer, enumeration, structure, or union type	An object of that type	Lvalue
Enumeration constant	The associated integer value	Not an lvalue

Primary expressions: Names

Name declared as	Evaluates to	Is an lvalue
Array	That array. In contexts subject to conversions, a pointer to the first object in the array, except where the name is used as the argument to the `sizeof` operator.	C Not an lvalue
Function	That function. In contexts subject to conversions, a pointer to that function, except where the name is used as the argument to the `sizeof` operator, or as the function in a function call expression.	C Not an lvalue C++ Lvalue

As an expression, a name may not refer to a label, `typedef` name, structure component name, union component name, structure tag, union tag, or enumeration tag. Names that can be referred to by a name in an expression reside in a name space that is separate from that of names for these purposes. Some of these names may be referred to within expressions by means of special constructs. For example, the dot or arrow operators may be used to refer to structure and union component names; `typedef` names may be used in casts or as an argument to the `sizeof` operator.

Literals

A literal is a numeric constant or string literal. When a literal is evaluated as an expression, its value is a constant. A lexical constant is never an lvalue. However, a string literal is an lvalue.

Related References
- “Literals” on page 21
- “The this Pointer” on page 269

Identifier Expressions

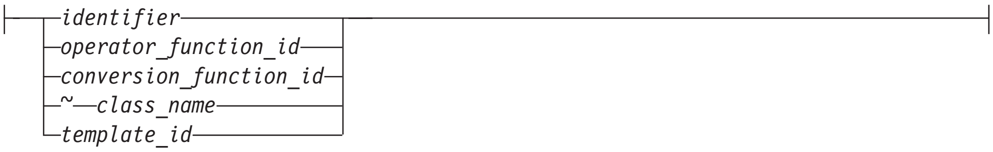
C++ An identifier expression, or *id-expression*, is a restricted form of primary expression. Syntactically, an *id-expression* requires a higher level of complexity than a simple identifier to provide a name for all of the language elements of C++.

An *id-expression* can be either a qualified or unqualified identifier. It can also appear after the dot and arrow operators.

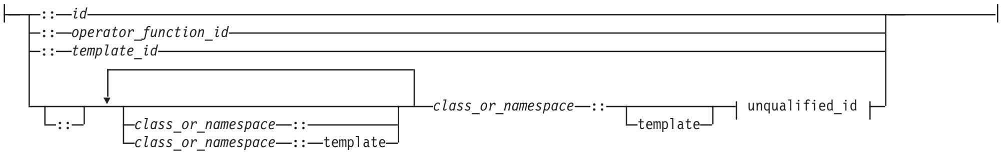
Syntax – id-expression

```
►►──┬─ unqualified_id ─┬──────────────────────────────────────────►◄
    └─ qualified_id ───┘
```

unqualified_id:



qualified_id:



Related References

- "Identifiers" on page 18
- Chapter 4, "Declarators," on page 81

Integer Constant Expressions

An *integer compile-time constant* is a value that is determined during compilation and cannot be changed at run time. An *integer compile-time constant expression* is an expression that is composed of constants and evaluated to a constant.

An integer constant expression is an expression that is composed of only the following:

- literals
- enumerators
- **const** variables
- **static** data members of integral or enumeration types
- casts to integral types
- **sizeof** expressions, where the operand is not a variable length array

▶ C The **sizeof** operator applied to a variable length array type is evaluated at run time, and therefore is not a constant expression.

You must use an integer constant expression in the following situations:

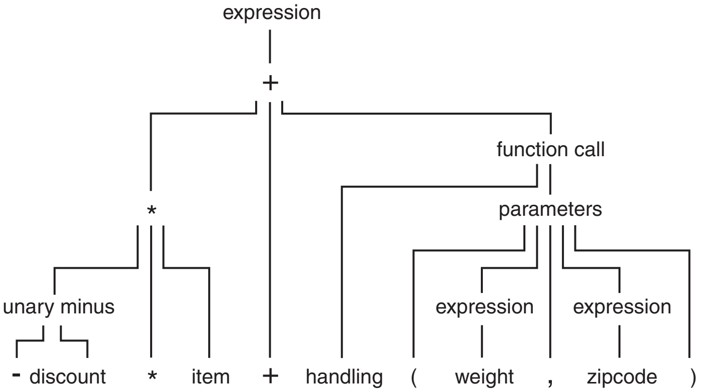
- In the subscript declarator as the description of an array bound.
- After the keyword **case** in a **switch** statement.
- In an enumerator, as the numeric value of an enum constant.
- In a bit-field width specifier.
- In the preprocessor **#if** statement. (Enumeration constants, address constants, and **sizeof** cannot be specified in a preprocessor **#if** statement.)

Related References

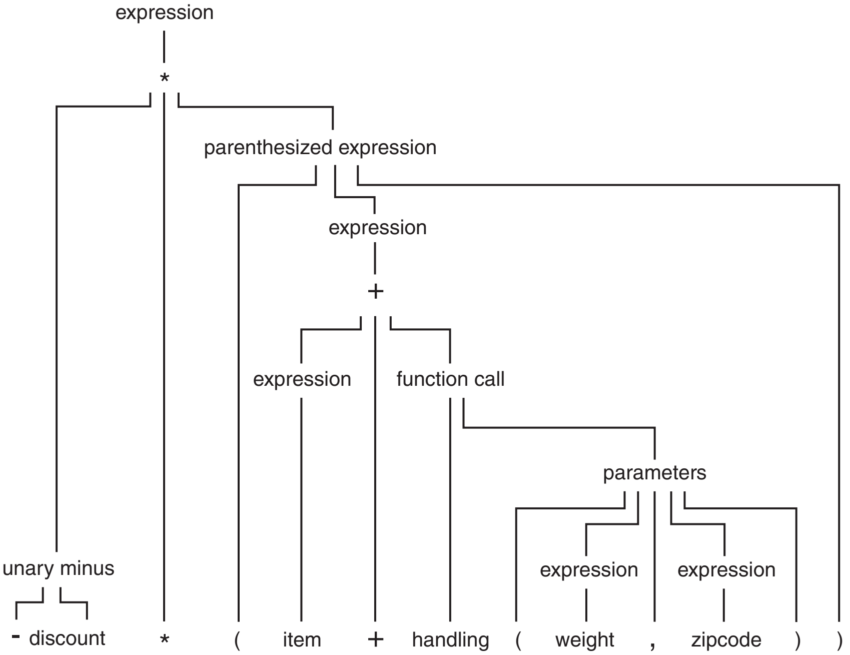
- "sizeof Operator" on page 124

Parenthesized Expressions ( )

Use parentheses to explicitly force the order of expression evaluation. The following expression does not use parentheses to group operands and operators. The parentheses surrounding `weight, zipcode` are used to form a function call. Note how the compiler groups the operands and operators in the expression according to the rules for operator precedence and associativity:



The following expression is similar to the previous expression, but it contains parentheses that change how the operands and operators are grouped:



In an expression that contains both associative and commutative operators, you can use parentheses to specify the grouping of operands with operators. The parentheses in the following expression guarantee the order of grouping operands with the operators:

```
x = f + (g + h);
```

C++ Scope Resolution Operator ::

C++ The `::` (scope resolution) operator is used to qualify hidden names so that you can still use them. You can use the unary scope operator if a namespace scope or global scope name is hidden by an explicit declaration of the same name in a block or class. For example:

```
int count = 0;

int main(void) {
  int count = 0;
  ::count = 1;  // set global count to 1
  count = 2;    // set local count to 2
  return 0;
}
```

The declaration of `count` declared in the `main()` function hides the integer named `count` declared in global namespace scope. The statement `::count = 1` accesses the variable named `count` declared in global namespace scope.

You can also use the class scope operator to qualify class names or class member names. If a class member name is hidden, you can use it by qualifying it with its class name and the class scope operator.

In the following example, the declaration of the variable `X` hides the class type `X`, but you can still use the static class member `count` by qualifying it with the class type `X` and the scope resolution operator.

```
#include <iostream>
using namespace std;

class X
{
public:
      static int count;
};
int X::count = 10;                // define static data member

int main ()
{
      int X = 0;                  // hides class type X
      cout << X::count << endl;   // use static member of class X
}
```

Related References

- "Scope of Class Names" on page 256
- Chapter 10, "Namespaces," on page 229

Postfix Expressions

Postfix operators are operators that appear after their operands. A *postfix expression* is a primary expression, or a primary expression that contains a postfix operator. The following summarizes the available postfix operators:

Precedence and associativity of postfix operators

Rank	Right Associative?	Operator Function	Usage
2		member selection	*object* **.** *member*
2		member selection	*pointer* **->** *member*
2		subscripting	*pointer* **[** *expr* **]**
2		function call	*expr* **(** *expr_list* **)**
2		value construction	*type* **(** *expr_list* **)**
2		postfix increment	*lvalue* **++**
2		postfix decrement	*lvalue* **--**
2		compound literals	**(***type-name***)** **{***initializer-list***}**
2	yes	▶ C++ type identification	**typeid** **(** *type* **)**

Precedence and associativity of postfix operators

Rank	Right Associative?	Operator Function	Usage
2	yes	C++ type identification at run time	**typeid (** *expr* **)**
2	yes	C++ conversion checked at compile time	**static_cast <** *type* **> (** *expr* **)**
2	yes	C++ conversion checked at run time	**dynamic_cast <** *type* **> (** *expr* **)**
2	yes	C++ unchecked conversion	**reinterpret_cast <** *type* **> (** *expr* **)**
2	yes	C++ **const** conversion	**const_cast <** *type* **> (** *expr* **)**

Function Call Operator ( )

A *function call* is an expression containing a simple type name and a parenthesized argument list. The argument list can contain any number of expressions separated by commas. It can also be empty.

For example:

```
stub()
overdue(account, date, amount)
notify(name, date + 5)
report(error, time, date, ++num)
```

There are two kinds of function calls: ordinary function calls and C++ member function calls. Any function may call itself except for the function `main`.

Type of a Function Call

The type of a function call expression is the return type of the function. This type can either be a complete type, a reference type, or the type **void**. A function call is an lvalue if and only if the type of the function is a reference.

Arguments and Parameters

A *function argument* is an expression that you use within the parentheses of a function call. A *function parameter* is an object or reference declared within the parentheses of a function declaration or definition. When you call a function, the arguments are evaluated, and each parameter is initialized with the value of the corresponding argument. The semantics of argument passing are identical to those of assignment.

A function can change the values of its non-const parameters, but these changes have no effect on the argument unless the parameter is a reference type.

Linkage and Function Calls

C In C, if a function definition has external linkage and a return type of **int**, calls to the function can be made before it is explicitly declared because an implicit declaration of `extern int func();` is assumed. This is *not* true for C++.

Type Conversions of Arguments

Arguments that are arrays or functions are converted to pointers before being passed as function arguments.

Arguments passed to nonprototyped C functions undergo conversions: type **short** or **char** parameters are converted to **int**, and **float** parameters to **double**. Use a cast expression for other conversions.

The compiler compares the data types provided by the calling function with the data types that the called function expects and performs necessary type conversions. For example, when function funct is called, argument f is converted to a **double**, and argument c is converted to an **int**:

```
char * funct (double d, int i);
     /* ... */
int main(void)
{
   float f;
   char c;
   funct(f, c) /* f is converted to a double, c is converted to an int */
   return 0;
}
```

Evaluation Order of Arguments

The order in which arguments are evaluated is not specified. Avoid such calls as:

```
method(sample1, batch.process--, batch.process);
```

In this example, batch.process-- might be evaluated last, causing the last two arguments to be passed with the same value.

Example of Function Calls

In the following example, main passes func two values: 5 and 7. The function func receives copies of these values and accesses them by the identifiers: a and b. The function func changes the value of a. When control passes back to main, the actual values of x and y are not changed. The called function func only receives copies of the values of x and y, not the variables themselves.

```
/**
 ** This example illustrates function calls
 **/

#include <stdio.h>

void func (int a, int b)
{
   a += b;
   printf("In func, a = %d    b = %d\n", a, b);
}

int main(void)
{
   int x = 5, y = 7;
   func(x, y);
   printf("In main, x = %d    y = %d\n", x, y);
   return 0;
}
```

This program produces the following output:

```
In func, a = 12   b = 7
In main, x = 5    y = 7
```

Array Subscripting Operator [ ]

A postfix expression followed by an expression in [ ] (brackets) specifies an element of an array. The expression within the brackets is referred to as a *subscript*. The first element of an array has the subscript zero.

By definition, the expression a[b] is equivalent to the expression *((a) + (b)), and, because addition is associative, it is also equivalent to b[a]. Between expressions a and b, one must be a pointer to a type T, and the other must have integral or enumeration type. The result of an array subscript is an lvalue. The following example demonstrates this:

```
#include <stdio.h>

int main(void) {
  int a[3] = { 10, 20, 30 };
  printf("a[0] = %d\n", a[0]);
  printf("a[1] = %d\n", 1[a]);
  printf("a[2] = %d\n", *(2 + a));
  return 0;
}
```

The following is the output of the above example:

```
a[0] = 10
a[1] = 20
a[2] = 30
```

C C99 allows array subscripting on arrays that are not lvalues. However, using the address of a non-lvalue as an array subscript is still not allowed. The following example is valid in C99, but not in C89:

```
struct trio{int a[3];};
struct trio f();
foo (int index)
{
   return f().a[index];
}
```

C++ The above restrictions on the types of expressions required by the subscript operator, as well as the relationship between the subscript operator and pointer arithmetic, do not apply if you overload **operator[]** of a class.

The first element of each array has the subscript 0. The expression contract[35] refers to the 36th element in the array contract.

In a multidimensional array, you can reference each element (in the order of increasing storage locations) by incrementing the right-most subscript most frequently.

For example, the following statement gives the value 100 to each element in the array code[4][3][6]:

```
for (first = 0; first < 4; ++first)
   {
   for (second = 0; second < 3; ++second)
      {
      for (third = 0; third < 6; ++third)
         {
         code[first][second][third] =
            100;
          }
        }
      }
```

Dot Operator .

The **.** (dot) operator is used to access class, structure, or union members. The member is specified by a postfix expression, followed by a **.** (dot) operator, followed by a possibly qualified identifier or a pseudo-destructor name. The postfix expression must be an object of type **class**, **struct** or **union**. The name must be a member of that object.

The value of the expression is the value of the selected member. If the postfix expression and the name are lvalues, the expression value is also an lvalue. If the postfix expression is type-qualified, the same type qualifiers will apply to the designated member in the resulting expression.

Pseudo-destructors

C++ A *pseudo-destructor* is a destructor of a nonclass type named *type_name* in the following syntax diagram :

```
►►──┬─┬────┬──┬────────────────────────┬─type_name──::──~──type_name──────────┬──────────►◄
    │ └─::─┘  └─nested_name_specifier─┘                                       │
    ├─┬────┬──nested_name_specifier──template──template_identifier──::──~──type_name─┤
    │ └─::─┘                                                                  │
    └─┬────┬──┬────────────────────────┬─~──type_name─────────────────────────┘
      └─::─┘  └─nested_name_specifier─┘
```

Arrow Operator –>

The **->** (arrow) operator is used to access class, structure or union members using a pointer. A postfix expression, followed by an **->** (arrow) operator, followed by a possibly qualified identifier or a pseudo-destructor name, designates a member of the object to which the pointer points. (A *pseudo-destructor* is a destructor of a nonclass type.) The postfix expression must be a pointer to an object of type **class**, **struct** or **union**. The name must be a member of that object.

The value of the expression is the value of the selected member. If the name is an lvalue, the expression value is also an lvalue. If the expression is a pointer to a qualified type, the same type-qualifiers will apply to the designated member in the resulting expression.

Related References

- "Dot Operator ."

The typeid Operator

C++ The **typeid** operator provides a program with the ability to retrieve the actual derived type of the object referred to by a pointer or a reference. This operator, along with the dynamic_cast operator, are provided for run-time type identification (RTTI) support in C++. The operator has the following form:

```
►►──typeid──(──┬─expr──────┬──)─────────────────────────────────────────►◄
               └─type-name─┘
```

The **typeid** operator requires run-time type information (RTTI) to be generated, which must be explicitly specified at compile time through a compiler option.

The **typeid** operator returns an lvalue of type **const std::type_info** that represents the type of expression *expr*. You must include the standard template library header **<typeinfo>** to use the typeid operator.

If *expr* is a reference or a dereferenced pointer to a polymorphic class, **typeid** will return a **type_info** object that represents the object that the reference or pointer denotes at run time. If it is not a polymorphic class, **typeid** will return a **type_info** object that represents the type of the reference or dereferenced pointer. The following example demonstrates this:

```
#include <iostream>
#include <typeinfo>
using namespace std;

struct A { virtual ~A() { } };
struct B : A { };

struct C { };
struct D : C { };

int main() {
  B bobj;
  A* ap = &bobj;
  A& ar = bobj;
  cout << "ap: " << typeid(*ap).name() << endl;
  cout << "ar: " << typeid(ar).name() << endl;

  D dobj;
  C* cp = &dobj;
  C& cr = dobj;
  cout << "cp: " << typeid(*cp).name() << endl;
  cout << "cr: " << typeid(cr).name() << endl;
}
```

The following is the output of the above example:

```
ap: B
ar: B
cp: C
cr: C
```

Classes `A` and `B` are polymorphic; classes `C` and `D` are not. Although `cp` and `cr` refer to an object of type `D`, `typeid(*cp)` and `typeid(cr)` return objects that represent class `C`.

Lvalue-to-rvalue, array-to-pointer, and function-to-pointer conversions will not be applied to *expr*. For example, the output of the following example will be `int [10]`, not `int *`:

```
#include <iostream>
#include <typeinfo>
using namespace std;

int main() {
  int myArray[10];
  cout << typeid(myArray).name() << endl;
}
```

If *expr* is a class type, that class must be completely defined.

The **typeid** operator ignores top-level **const** or **volatile** qualifiers.

static_cast Operator

C++ The *static_cast operator* converts a given expression to a specified type.

Syntax – static_cast

```
►►──static_cast──<──Type──>──(──expression──)────────────────────────────►◄
```

The following is an example of the **static_cast** operator.

```
#include <iostream>
using namespace std;

int main() {
  int j = 41;
  int v = 4;
  float m = j/v;
  float d = static_cast<float>(j)/v;
  cout << "m = " << m << endl;
  cout << "d = " << d << endl;
}
```

The following is the output of the above example:

```
m = 10
d = 10.25
```

In this example, `m = j/v;` produces an answer of type int because both `j` and `v` are integers. Conversely, `d = static_cast<float>(j)/v;` produces an answer of type float. The **static_cast** operator converts variable `j` to a type **float**. This allows the compiler to generate a division with an answer of type **float**. All **static_cast** operators resolve at compile time and do not remove any **const** or **volatile** modifiers.

Applying the **static_cast** operator to a null pointer will convert it to a null pointer value of the target type.

You can explicitly convert a pointer of a type `A` to a pointer of a type `B` if `A` is a base class of `B`. If `A` is not a base class of `B`, a compiler error will result.

You may cast an lvalue of a type `A` to a type `B&` if the following are true:

- `A` is a base class of `B`
- You are able to convert a pointer of type `A` to a pointer of type `B`
- The type `B` has the same or greater **const** or **volatile** qualifiers than type `A`
- `A` is not a virtual base class of `B`

The result is an lvalue of type `B`.

A pointer to member type can be explicitly converted into a different pointer to member type if both types are pointers to members of the same class. This form of explicit conversion may also take place if the pointer to member types are from separate classes, however one of the class types must be derived from the other.

reinterpret_cast Operator

C++ A *reinterpret_cast operator* handles conversions between unrelated types.

Syntax – reinterpret_cast

```
▶▶—reinterpret_cast—<—Type—>—(—expression—)————————————————————————▶◀
```

The reinterpret_cast operator produces a value of a new type that has the same bit pattern as its argument. You cannot cast away a **const** or **volatile** qualification. You can explicitly perform the following conversions:

- A pointer to any integral type large enough to hold it
- A value of integral or enumeration type to a pointer
- A pointer to a function to a pointer to a function of a different type
- A pointer to an object to a pointer to an object of a different type
- A pointer to a member to a pointer to a member of a different class or type, if the types of the members are both function types or object types

A null pointer value is converted to the null pointer value of the destination type.

Given an lvalue expression of type T and an object x, the following two conversions are synonymous:

- `reinterpret_cast<T&>(x)`
- `*reinterpret_cast<T*>(&x)`

ISO C++ also supports C-style casts. The two styles of explicit casts have different syntax but the same semantics, and either way of reinterpreting one type of pointer as an incompatible type of pointer is usually invalid. The reinterpret_cast operator, as well as the other named cast operators, is more easily spotted than C-style casts, and highlights the paradox of a strongly typed language that allows explicit casts.

The C++ compiler detects and quietly fixes most but not all violations. It is important to remember that even though a program compiles, its source code may not be completely correct. On some platforms, performance optimizations are predicated on strict adherence to ISO aliasing rules. Although the C++ compiler tries to help with type-based aliasing violations, it cannot detect all possible cases.

The following example violates the aliasing rule, but will execute as expected when compiled unoptimized in C++ or in K&R C. It will also successfully compile optimized in C++, but will not necessarily execute as expected. The offending line 7 causes an old or uninitialized value for x to be printed.

```
1  extern int y = 7.;
2
3  int main() {
4       float x;
5       int i;
6       x = y;
7       i = *(int *) &x;
8       printf("i=%d. x=%f.\n", i, x);
9  }
```

The next code example contains an incorrect cast that the compiler cannot even detect because the cast is across two different files.

```
/* separately compiled file 1 */
    extern float f;
    extern int * int_pointer_to_f = (int *) &f; /* suspicious cast */

/* separately compiled file 2 */
    extern float f;
```

```
7      extern int * int_pointer_to_f;
8      f = 1.0;
9      int i = *int_pointer_to_f;           /* no suspicious cast but wrong */
```

In line 8, there is no way for the compiler to know that `f = 1.0` is storing into the same object that `int i = *int_pointer_to_f` is loading from.

Related References

- "Standard Type Conversions" on page 150
- "User-Defined Conversions" on page 328

const_cast Operator

C++ A *const_cast operator* is used to add or remove a **const** or **volatile** modifier to or from a type.

Syntax – const_cast

```
▶▶—const_cast—<—Type—>—(—expression—)——————————————————————————————▶◀
```

Type and the type of *expression* may only differ with respect to their **const** and **volatile** qualifiers. Their cast is resolved at compile time. A single **const_cast** expression may add or remove any number of **const** or **volatile** modifiers.

The result of a **const_cast** expression is an rvalue unless *Type* is a reference type. In this case, the result is an lvalue.

Types can not be defined within **const_cast**.

The following demonstrates the use of the **const_cast** operator:

```
#include <iostream>
using namespace std;

void f(int* p) {
  cout << *p << endl;
}

int main(void) {
  const int a = 10;
  const int* b = &a;

  // Function f() expects int*, not const int*
  //   f(b);
  int* c = const_cast<int>(b);
  f(c);

  // Lvalue is const
  //  *b = 20;

  // Undefined behavior
  //  *c = 30;

  int a1 = 40;
  const int* b1 = &a1;
  int* c1 = const_cast<int>(b1);

  // Integer a1, the object referred to by c1, has
  // not been declared const
  *c1 = 50;

  return 0;
}
```

The compiler will not allow the function call f(b). Function f() expects a pointer to an **int**, not a **const int**. The statement int* c = const_cast<int>(b) returns a pointer c that refers to a without the **const** qualification of a. This process of using **const_cast** to remove the **const** qualification of an object is called *casting away constness*. Consequently the compiler will allow the function call f(c).

The compiler would not allow the assignment *b = 20 because b points to an object of type **const int**. The compiler will allow the *c = 30, but the behavior of this statement is undefined. If you cast away the constness of an object that has been explicitly declared as **const**, and attempt to modify it, the results are undefined.

However, if you cast away the constness of an object that has not been explicitly declared as **const**, you can modify it safely. In the above example, the object referred to by b1 has not been declared **const**, but you cannot modify this object through b1. You may cast away the constness of b1 and modify the value to which it refers.

Related References
- "Type Qualifiers" on page 71

dynamic_cast Operator

C++ The *dynamic_cast* operator performs type conversions at run time. The **dynamic_cast** operator guarantees the conversion of a pointer to a base class to a pointer to a derived class, or the conversion of an lvalue referring to a base class to a reference to a derived class. A program can thereby use a class hierarchy safely. This operator and the typeid operator provide run-time type information (RTTI) support in C++.

The expression dynamic_cast<T>(v) converts the expression v to type T. Type T must be a pointer or reference to a complete class type or a pointer to void. If T is a pointer and the **dynamic_cast** operator fails, the operator returns a null pointer of type T. If T is a reference and the **dynamic_cast** operator fails, the operator throws the exception **std::bad_cast**. You can find this class in the standard library header **<typeinfo>**.

The **dynamic_cast** operator requires run-time type information (RTTI) to be generated, which must be explicitly specified at compile time through a compiler option.

If T is a void pointer, then **dynamic_cast** will return the starting address of the object pointed to by v. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct A {
  virtual ~A() { };
};

struct B : A { };

int main() {
  B bobj;
  A* ap = &bobj;
  void * vp = dynamic_cast<void *>(ap);
  cout << "Address of vp  : " << vp << endl;
  cout << "Address of bobj: " << &bobj << endl;
}
```

The output of this example will be similar to the following. Both vp and &bobj will refer to the same address:

```
Address of vp  : 12FF6C
Address of bobj: 12FF6C
```

The primary purpose for the **dynamic_cast** operator is to perform type-safe *downcasts*. A downcast is the conversion of a pointer or reference to a class A to pointer or reference to a class B, where class A is a base class of B. The problem with downcasts is that a pointer of type A* can and must point to any object of a class that has been derived from A. The **dynamic_cast** operator ensures that if you convert a pointer of class A to a pointer of a class B, the object that A points to belongs to class B or a class derived from B.

The following example demonstrates the use of the **dynamic_cast** operator:

```
#include <iostream>
using namespace std;

struct A {
  virtual void f() { cout << "Class A" << endl; }
};

struct B : A {
  virtual void f() { cout << "Class B" << endl; }
};

struct C : A {
  virtual void f() { cout << "Class C" << endl; }
};

void f(A* arg) {
  B* bp = dynamic_cast<B*>(arg);
  C* cp = dynamic_cast<C*>(arg);

  if (bp)
    bp->f();
  else if (cp)
    cp->f();
  else
    arg->f();
};

int main() {
  A aobj;
  C cobj;
  A* ap = &cobj;
  A* ap2 = &aobj;
  f(ap);
  f(ap2);
}
```

The following is the output of the above example:

```
Class C
Class A
```

The function `f()` determines whether the pointer `arg` points to an object of type `A`, `B`, or `C`. The function does this by trying to convert `arg` to a pointer of type `B`, then to a pointer of type `C`, with the **dynamic_cast** operator. If the **dynamic_cast** operator succeeds, it returns a pointer that points to the object denoted by `arg`. If **dynamic_cast** fails, it returns `0`.

You may perform downcasts with the **dynamic_cast** operator only on polymorphic classes. In the above example, all the classes are polymorphic because class `A` has a virtual function. The **dynamic_cast** operator uses the run-time type information generated from polymorphic classes.

Related References
- "Derivation" on page 287
- "User-Defined Conversions" on page 328

Unary Expressions

A *unary expression* contains one operand and a unary operator. All unary operators have the same precedence and have right-to-left associativity. A unary expression is therefore a postfix expression.

As indicated in the following descriptions, the usual arithmetic conversions are performed on the operands of most unary expressions.

The following table summarizes the operators for unary expressions:

Precedence and associativity of unary operators

Rank	Right Associative?	Operator Function	Usage
3	yes	size of object in bytes	**sizeof (** *expr* **)**
3	yes	size of type in bytes	**sizeof** *type*
3	yes	prefix increment	**++** *lvalue*
3	yes	prefix decrement	**--** *lvalue*
3	yes	complement	**~** *expr*
3	yes	not	**!** *expr*
3	yes	unary minus	**-** *expr*
3	yes	unary plus	**+** *expr*
3	yes	address of	**&** *lvalue*
3	yes	indirection or dereference	***** *expr*
3	yes	C++ create (allocate memory)	**new** *type*
3	yes	C++ create (allocate and initialize memory)	**new** *type* **(** *expr_list* **)** *type*
3	yes	C++ create (placement)	**new** *type* **(** *expr_list* **)** *type* (*expr_list*)
3	yes	C++ destroy (deallocate memory)	**delete** *pointer*
3	yes	C++ destroy array	**delete [ ]** *pointer*
3	yes	type conversion (cast)	**(** *type* **)** *expr*

C C99 adds the unary operator `_Pragma`, which allows a preprocessor macro to contain a pragma directive. The operator is supported by IBM C++ as an orthogonal language extension.

XL C/C++ extends the C99 and C++ standards to support the unary operators `__real__` and `__imag__`. These operators provide the ability to extract the real and imaginary parts of a complex type. These extensions have been implemented to ease the porting applications developed with GNU C and C++.

Related References
- "Complex Literals" on page 26

Increment ++

The ++ (increment) operator adds 1 to the value of a scalar operand, or if the operand is a pointer, increments the operand by the size of the object to which it points. The operand receives the result of the increment operation. The operand must be a modifiable lvalue of arithmetic or pointer type.

You can put the ++ before or after the operand. If it appears before the operand, the operand is incremented. The incremented value is then used in the expression. If you put the ++ after the operand, the value of the operand is used in the expression *before* the operand is incremented. For example:

```
play = ++play1 + play2++;
```

is similar to the following expressions; `play2` is altered before `play`:

```
int temp, temp1, temp2;

temp1 = play1 + 1;
temp2 = play2;
play1 = temp1;
temp = temp1 + temp2;
play2 = play2 + 1;
play = temp;
```

The result has the same type as the operand after integral promotion.

The usual arithmetic conversions on the operand are performed.

Decrement −−

The `--` (decrement) operator subtracts 1 from the value of a scalar operand, or if the operand is a pointer, decreases the operand by the size of the object to which it points. The operand receives the result of the decrement operation. The operand must be a modifiable lvalue.

You can put the `--` before or after the operand. If it appears before the operand, the operand is decremented, and the decremented value is used in the expression. If the `--` appears after the operand, the current value of the operand is used in the expression and the operand is decremented.

For example:

```
play = --play1 + play2--;
```

is similar to the following expressions; `play2` is altered before `play`:

```
int temp, temp1, temp2;

temp1 = play1 - 1;
temp2 = play2;
play1 = temp1;
temp = temp1 + temp2;
play2 = play2 - 1;
play = temp;
```

The result has the same type as the operand after integral promotion, but is not an lvalue.

The usual arithmetic conversions are performed on the operand.

Unary Plus +

The + (unary plus) operator maintains the value of the operand. The operand can have any arithmetic type or pointer type. The result is not an lvalue.

The result has the same type as the operand after integral promotion.

Note: Any plus sign in front of a constant is not part of the constant.

Unary Minus –

The - (unary minus) operator negates the value of the operand. The operand can have any arithmetic type. The result is not an lvalue.

For example, if `quality` has the value `100`, `-quality` has the value `-100`.

The result has the same type as the operand after integral promotion.

Note: Any minus sign in front of a constant is not part of the constant.

Logical Negation !

The `!` (logical negation) operator determines whether the operand evaluates to `0` (false) or nonzero (true).

▶ C The expression yields the value `1` (true) if the operand evaluates to `0`, and yields the value `0` (false) if the operand evaluates to a nonzero value.

▶ C++ The expression yields the value **true** if the operand evaluates to false (0), and yields the value **false** if the operand evaluates to true (nonzero). The operand is implicitly converted to **bool**, and the type of the result is **bool**.

The following two expressions are equivalent:

```
!right;
right == 0;
```

Bitwise Negation ~

The ~ (bitwise negation) operator yields the bitwise complement of the operand. In the binary representation of the result, every bit has the opposite value of the same bit in the binary representation of the operand. The operand must have an integral type. The result has the same type as the operand but is not an lvalue.

Suppose `x` represents the decimal value `5`. The 16-bit binary representation of `x` is:

```
0000000000000101
```

The expression ~`x` yields the following result (represented here as a 16-bit binary number):

```
1111111111111010
```

Note that the ~ character can be represented by the trigraph `??-`.

The 16-bit binary representation of ~0 is:

```
1111111111111111
```

Address &

The & (address) operator yields a pointer to its operand. The operand must be an lvalue, a function designator, or a qualified name. It cannot be a bit field, nor can it have the storage class **register**.

If the operand is an lvalue or function, the resulting type is a pointer to the expression type. For example, if the expression has type **int**, the result is a pointer to an object having type **int**.

If the operand is a qualified name and the member is not static, the result is a pointer to a member of class and has the same type as the member. The result is not an lvalue.

If p_to_y is defined as a pointer to an **int** and y as an **int**, the following expression assigns the address of the variable y to the pointer p_to_y :

```
p_to_y = &y;
```

▶ Linux ▶ Mac OS X The address operator has been extended to handle vector types, provided that AltiVec language extensions have been enabled. The result of the address operator applied to a vector type can be stored in a pointer to a compatible vector type. The address of a vector type can be used to initialize a pointer to vector type if both sides of the initialization have compatible types. A pointer to **void** can also be initialized with the address of a vector type.

▶ C++ The remainder of this section pertains to C++ only.

The ampersand symbol & is used in C++ as a reference declarator in addition to being the address operator. The meanings are related but not identical.

```
int target;
int &rTarg = target;  // rTarg is a reference to an integer.
                      // The reference is initialized to refer to target.
void f(int*& p);      // p is a reference to a pointer
```

If you take the address of a reference, it returns the address of its target. Using the previous declarations, &rTarg is the same memory address as &target.

You may take the address of a register variable.

You can use the & operator with overloaded functions only in an initialization or assignment where the left side uniquely determines which version of the overloaded function is used.

The address of a label can be taken using the GNU C address operator &&. The label can thus be used as a value.

Related References

- “Pointers” on page 83
- “References” on page 96
- “Labels as Values” on page 190

Indirection *

The * (indirection) operator determines the value referred to by the pointer-type operand. The operand cannot be a pointer to an incomplete type. If the operand points to an object, the operation yields an lvalue referring to that object. If the operand points to a function, the result is a function designator in C or, in C++, an lvalue referring to the object to which the operand points. Arrays and functions are converted to pointers.

The type of the operand determines the type of the result. For example, if the operand is a pointer to an **int**, the result has type **int**.

Do not apply the indirection operator to any pointer that contains an address that is not valid, such as `NULL`. The result is not defined.

If `p_to_y` is defined as a pointer to an **int** and `y` as an **int**, the expressions:

```
p_to_y = &y;
*p_to_y = 3;
```

cause the variable `y` to receive the value `3`.

▶ Linux ▶ Mac OS X The indirection operator has been extended to handle vector types, provided that AltiVec language extensions have been enabled.

Related References

- “Pointers” on page 83

alignof Operator

The **__alignof__** operator returns the number of bytes used in the alignment of its operand. The language feature is orthogonal to C89 and C99. The operand can be an expression or a parenthesized type identifier. If the operand is an expression representing an lvalue, the number returned by **__alignof__** represents the alignment that the lvalue is known to have. The type of the expression is determined at compile time, but the expression itself is not evaluated. If the operand is a type, the number represents the alignment usually required for the type on the target platform.

The **__alignof__** operator may not be applied to the following:

- An lvalue representing a bit field
- A function type
- An undefined structure or class
- An incomplete type (such as **void**)

An **__alignof__** expression has the form:

```
►►──__alignof__──┬─unary_expression─┬──────────────────────────────────────►◄
                 └─(──type-id──)────┘
```

If *type-id* is a reference or a referenced type, the result is the alignment of the referenced type. If *type-id* is an array, the result is the alignment of the array element type. If *type-id* is a fundamental type, the result is implementation-defined.

For example, on Linux, `__alignof__(long)` returns `4` in 32-bit mode, and `8` in 64-bit mode.

The operand of **__alignof__** can be a vector type, provided that the AltiVec language extensions are enabled. For example,

```
vector unsigned int v1 = (vector unsigned int)(10);
vector unsigned int *pv1 = &v1;
__alignof__(v1); // vector type alignment: 16.
__alignof__(&v1); // address of vector alignment: 4.
__alignof__(*pv1); // dereferenced pointer to vector alignment: 16.
__alignof__(pv1); // pointer to vector alignment: 4.
__alignof__(vector signed char); // vector type alignment: 16.
```

When **__attribute__((aligned))** is used to increase the alignment of a variable of vector type, the value returned by the **__alignof__** operator is the alignment factor specified by **__attribute__((aligned))**.

Related References
- “The aligned Variable Attribute” on page 32

sizeof Operator

The **sizeof** operator yields the size in *bytes* of the operand, which can be an expression or the parenthesized name of a type. A **sizeof** expression has the form:

```
►►──sizeof──┬─expr──────────────┬──────────────────────────────────────►◄
            └─(──type-name──)───┘
```

The result for either kind of operand is not an lvalue, but a constant integer value. The type of the result is the unsigned integral type **size_t** defined in the header file `stddef.h`.

The **sizeof** operator applied to a type name yields the amount of memory that would be used by an object of that type, including any internal or trailing padding. The size of any of the three kinds of **char** objects (**unsigned**, **signed**, or plain) is the size of a byte, 1. If the operand is a variable length array type, the operand is evaluated. The **sizeof** operator may not be applied to:
- A bit field
- A function type
- An undefined structure or class
- An incomplete type (such as **void**)

The **sizeof** operator applied to an expression yields the same result as if it had been applied to only the name of the type of the expression. At compile time, the compiler analyzes the expression to determine its type, but does not evaluate it. None of the usual type conversions that occur in the type analysis of the expression are directly attributable to the **sizeof** operator. However, if the operand contains operators that perform conversions, the compiler does take these conversions into consideration in determining the type.

The second line of the following sample causes the usual arithmetic conversions to be performed. Assuming that a **short** uses 2 bytes of storage and an **int** uses 4 bytes,

```
short x; ... sizeof (x)        /* the value of sizeof operator is 2 */
short x; ... sizeof (x + 1)    /* value is 4, result of addition is type int */
```

The result of the expression `x + 1` has type **int** and is equivalent to `sizeof(int)`. The value is also `4` if `x` has type **char**, **short**, or **int** or any enumeration type.

Types cannot be defined in a **sizeof** expression.

In the following example, the compiler is able to evaluate the size at compile time. The operand of **sizeof**, an expression, is not evaluated. The value of `b` is the integer constant 5, from initialization to the end of program run time:

```
#include <stdio.h>

int main(void){
  int b = 5;
  sizeof(b++);
  return 0;
}
```

Except in preprocessor directives, you can use a **sizeof** expression wherever an integral constant is required. One of the most common uses for the **sizeof** operator is to determine the size of objects that are referred to during storage allocation, input, and output functions.

Another use of **sizeof** is in porting code across platforms. You should use the **sizeof** operator to determine the size that a data type represents. For example:

```
sizeof(int);
```

▶ Linux ▶ Mac OS X ▶ C The operand of the **sizeof** operator can be a vector type or the result of dereferencing a pointer to vector type, provided that the AltiVec language extensions have been enabled. In these cases, the return value of **sizeof** is always 16.

```
vector bool int v1;
vector bool int *pv1 = &v1;
sizeof(v1); // vector type: 16.
sizeof(&v1); // address of vector: 4.
sizeof(*pv1); // dereferenced pointer to vector: 16.
sizeof(pv1); // pointer to vector: 4.
sizeof(vector bool int); // vector type: 16.
```

The result of a sizeof expression depends on the type it is applied to.

Operand	Result
An array	The result is the total number of bytes in the array. For example, in an array with 10 elements, the size is equal to 10 times the size of a single element. The compiler does not convert the array to a pointer before evaluating the expression.
▶ C++ A class	The result is always nonzero, and is equal to the number of bytes in an object of that class including any padding required for placing class objects in an array.
▶ C++ A reference	The result is the size of the referenced object.

typeof Operator

The **typeof** operator returns the type of its argument, which can be an expression or a type. The language feature provides a way to derive the type from an expression. The alternate spelling of the keyword, **__typeof__**, is recommended. Given an expression `e`, `__typeof__(e)` can be used anywhere a type name is needed, for example in a declaration or in a cast.

The **typeof** operator is an orthogonal language extension provided for handling programs developed with GNU C. The language feature has been extended for the Mac OS X platform to accept a vector type as its operand, when AltiVec language extensions have been enabled.

A **typeof** construct is of the form:

```
►►──┬─__typeof__─┬──(──┬─expr──────┬──)─────────────────────────────────►◄
    └─typeof─────┘     └─type-name─┘
```

A **typeof** construct itself is not an expression, but the name of a type. A **typeof** construct behaves like a type name defined using **typedef**, although the syntax resembles that of **sizeof**.

The following examples illustrate its basic syntax. For an expression `e`:

```
int e;
__typeof__(e + 1) j;   /* the same as declaring int j;     */
e = (__typeof__(e)) f; /* the same as casting e = (int) f; */
```

Using a **typeof** construct is equivalent to declaring a `typedef` name. Given

```
int T[2];
int i[2];
```

you can write

```
__typeof__(i) a;        /* all three constructs have the same meaning */
__typeof__(int[2]) a;
__typeof__(T) a;
```

The behavior of the code is as if you had declared `int a[2];`.

For a bit field, **typeof** represents the underlying type of the bit field. For example, `int m:2;`, the `typeof(m)` is **int**. Since the bit field property is not reserved, `n` in `typeof(m) n;` is the same as `int n`, but not `int n:2`.

The **typeof** operator can be nested inside `sizeof` and itself. The following declarations of `arr` as an array of pointers to **int** are equivalent:

```
int *arr[10];                     /* traditional C declaration          */
__typeof__(__typeof__ (int *)[10]) a;  /* equivalent declaration  */
```

The **typeof** operator can be useful in macro definitions where expression `e` is a parameter. For example,

```
#define SWAP(a,b) { __typeof__(a) temp; temp = a; a = b; b = temp; }
```

Label Value Operator &&

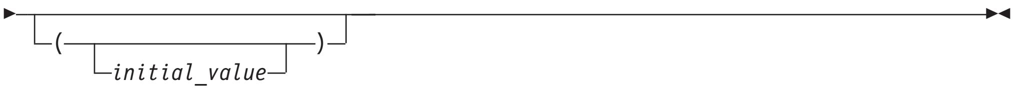
The label value operator **&&** returns the address of its operand, which must be a label defined in the current function or a containing function. The value is a constant of type **void*** and should be used only in a computed goto statement. The language feature is an orthogonal extension to C and C++, implemented to facilitate porting programs developed with GNU C.

Related References
- "Labels as Values" on page 190
- "Computed goto" on page 207

C++ new Operator

C++ The **new** operator provides dynamic storage allocation. The syntax for an allocation expression containing the **new** operator is:

```
►►──┬────┬──new──┬─────────────────────┬──┬─(─type─)─┬──────────────────►
    └─::─┘       └─(─argument_list─)───┘  └─new_type─┘
```



If you prefix **new** with the scope resolution operator (`::`), the global **operator new()** is used. If you specify an *argument_list*, the overloaded **new** operator that corresponds to that *argument_list* is used. The *type* is an existing built-in or user-defined type. A *new_type* is a type that has not already been defined and can include type specifiers and declarators.

An allocation expression containing the **new** operator is used to find storage in free store for the object being created. The *new expression* returns a pointer to the object created and can be used to initialize the object. If the object is an array, a pointer to the initial element is returned.

You can use `set_new_handler()` only to specify what **new** does when it fails.

You cannot use the **new** operator to allocate function types, **void**, or incomplete class types because these are not object types. However, you can allocate pointers to functions with the **new** operator. You cannot create a reference with the **new** operator.

When the object being created is an array, only the first dimension can be a general expression. All subsequent dimensions must be constant integral expressions. The first dimension can be a general expression even when an existing *type* is used. You can create an array with zero bounds with the **new** operator. For example:

```
char * c = new char[0];
```

In this case, a pointer to a unique object is returned.

An object created with **operator new()** or **operator new[]()** exists until the **operator delete()** or **operator delete[]()** is called to deallocate the object's memory. A **delete** operator or a destructor will not be implicitly called for an object created with a **new** that has not been explicitly deallocated before the end of the program.

If parentheses are used within a new type, parentheses should also surround the new type to prevent syntax errors.

In the following example, storage is allocated for an array of pointers to functions:

```
void f();
void g();

int main(void)
{
      void (**p)(), (**q)();
      // declare p and q as pointers to pointers to void functions
      p = new (void (*[3])());
      // p now points to an array of pointers to functions
      q = new void(*[3])(); // error
      // error - bound as 'q = (new void) (*[3])();'
      p[0] = f;  // p[0] to point to function f
      q[2] = g;  // q[2] to point to function g
      p[0]();    // call f()
      q[2]();    // call g()
      return (0);
}
```

However, the second use of **new** causes an erroneous binding of `q = (new void) (*[3])()`.

The type of the object being created cannot contain class declarations, enumeration declarations, or **const** or **volatile** types. It can contain pointers to **const** or **volatile** objects.

For example, `const char*` is allowed, but `char* const` is not.

Placement Syntax

Additional arguments can be supplied to **new** by using the *argument_list*, also called the *placement syntax*. If placement arguments are used, a declaration of **operator new()** or **operator new[]()** with these arguments must exist. For example:

```
#include <new>
using namespace std;

class X
{
public:
      void* operator new(size_t,int, int){ /* ... */ }
};

// ...

int main ()
{
      X* ptr = new(1,2) X;
}
```

The placement syntax is commonly used to invoke the global placement `new` function. The global placement `new` function initializes an object or objects at the location specified by the placement argument in the placement new expression. This location must address storage that has previously been allocated by some other means, because the global placement `new` function does not itself allocate memory. In the following example, no new memory is allocated by the calls `new(whole) X(8);`, `new(seg2) X(9);`, or `new(seg3) X(10);` Instead, the constructors `X(8)`, `X(9)`, and `X(10)` are called to reinitialize the memory allocated to the buffer `whole`.

Because placement `new` does not allocate memory, you should not use `delete` to deallocate objects created with the placement syntax. You can only delete the entire memory pool (`delete whole`). In the example, you can keep the memory buffer but destroy the object stored in it by explicitly calling a destructor.

```
#include <new>
class X
{
   public:
      X(int n): id(n){ }
      ~X(){ }
   private:
      int id;
      // ...
};

int main()
{
   char* whole = new char[ 3 * sizeof(X) ];  // a 3-part buffer
   X * p1 = new(whole) X(8);                 // fill the front
   char* seg2 = &whole[ sizeof(X) ];         // mark second segment
   X * p2 = new(seg2) X(9);                  // fill second segment
   char* seg3 = &whole[ 2 * sizeof(X) ];     // mark third segment
   X * p3 = new(seg3) X(10);                 // fill third segment

   p2->~X();   // clear only middle segment, but keep the buffer
   // ...
   return 0;
}
```

The placement new syntax can also be used for passing parameters to an allocation routine rather than to a constructor.

Related References
- “Free Store” on page 323
- “set_new_handler() — Set Behavior for new Failure”
- “C++ delete Operator” on page 130
- “C++ Scope Resolution Operator ::” on page 107
- “Constructors and Destructors Overview” on page 311
- “Objects” on page 35

Initializing Objects Created with the new Operator

C++ You can initialize objects created with the **new** operator in several ways. For nonclass objects, or for class objects without constructors, a *new initializer* expression can be provided in a new expression by specifying (*expression*) or (). For example:

```
double* pi = new double(3.1415926);
int* score = new int(89);
float* unknown = new float();
```

If a class does not have a default constructor, the new initializer must be provided when any object of that class is allocated. The arguments of the new initializer must match the arguments of a constructor.

You cannot specify an initializer for arrays. You can initialize an array of class objects only if the class has a default constructor. The constructor is called to initialize each array element (class object).

Initialization using the new initializer is performed only if **new** successfully allocates storage.

Related References
- “Free Store” on page 323
- “Constructors and Destructors Overview” on page 311

set_new_handler() — Set Behavior for new Failure

C++ When the **new** operator creates a new object, it calls the **operator new()** or **operator new[]()** function to obtain the needed storage.

When **new** cannot allocate storage to create a new object, it calls a *new handler* function if one has been installed by a call to set_new_handler(). The std::set_new_handler() function is declared in the header **<new>**. Use it to call a new handler you have defined or the default new handler.

Your new handler must perform one of the following:
- obtain more storage for memory allocation, then return
- throw an exception of type std::bad_alloc or a class derived from std::bad_alloc
- call either abort() or exit()

The `set_new_handler()` function has the prototype:

```
typedef void(*PNH)();
PNH set_new_handler(PNH);
```

`set_new_handler()` takes as an argument a pointer to a function (the new handler), which has no arguments and returns void. It returns a pointer to the previous new handler function.

If you do not specify your own `set_new_handler()` function, **new** throws an exception of type `std::bad_alloc`.

The following program fragment shows how you could use `set_new_handler()` to return a message if the **new** operator cannot allocate storage:

```
#include <iostream>
#include <new>
#include <cstdlib>
using namespace std;

void no_storage()
{
    std::cerr << "Operator new failed: no storage is
    available.\n";
      std::exit(1);
}
int main(void)
{
     std::set_new_handler(&no_storage);
   // Rest of program ...
}
```

If the program fails because **new** cannot allocate storage, the program exits with the message:

```
 Operator new failed:
no storage is available.
```

Related References
- "C++ new Operator" on page 126
- "Free Store" on page 323

C++ delete Operator

C++ The **delete** operator destroys the object created with **new** by deallocating the memory associated with the object.

The **delete** operator has a **void** return type. It has the syntax:

```
►►──┬────┬──delete──object_pointer──────────────────────────────────►◄
    └─::─┘
```

The operand of **delete** must be a pointer returned by **new**, and cannot be a pointer to constant. Deleting a null pointer has no effect.

The **delete[]** operator frees storage allocated for array objects created with **new[]**. The **delete** operator frees storage allocated for individual objects created with **new**.

It has the syntax:

```
►►──┬────┬──delete──[──]──array──────────────────────────────────────►◄
    └─::─┘
```

The result of deleting an array object with **delete** is undefined, as is deleting an individual object with **delete[]**. The array dimensions do not need to be specified with **delete[]**.

The result of any attempt to access a deleted object or array is undefined.

If a destructor has been defined for a class, **delete** invokes that destructor. Whether a destructor exists or not, **delete** frees the storage pointed to by calling the function **operator delete()** of the class if one exists.

The global **::operator delete()** is used if:
- The class has no **operator delete()**.
- The object is of a nonclass type.
- The object is deleted with the **::delete** expression.

The global **::operator delete[]()** is used if:
- The class has no **operator delete[]()**
- The object is of a nonclass type
- The object is deleted with the **::delete[]** expression.

The default global **operator delete()** only frees storage allocated by the default global **operator new()**. The default global **operator delete[]()** only frees storage allocated for arrays by the default global **operator new[]()**.

Related References
- "Free Store" on page 323
- "Constructors and Destructors Overview" on page 311
- "void Type" on page 53

Cast Expressions

The cast operator is used for *explicit type conversions*. This operator has the following form, where *T* is a type, and *expr* is an expression:

(*T* **)** *expr*

It converts the value of *expr* to the type *T*. In C, the result of this operation is not an lvalue. In C++, the result of this operation is an lvalue if *T* is a reference; in all other cases, the result is an rvalue.

▶ C++ The remainder of this section pertains to C++ only.

A cast is a valid lvalue if its operand is an lvalue. In the following simple assignment expression, the right-hand side is first converted to the specified type, then to the type of the inner left-hand side expression, and the result is stored. The value is converted back to the specified type, and becomes the value of the assignment. In the following example, `i` is of type `char *`.

```
(int)i = 8     // This is equivalent to the following expression
(int)(i = (char*) (int)(8))
```

For compound assignment operation applied to a cast, the arithmetic operator of the compound assignment is performed using the type resulting from the cast, and then proceeds as in the case of simple assignment. The following expressions are equivalent. Again, `i` is of type `char *`.

```
(int)i += 8     // This is equivalent to the following expression
(int)(i = (char*) (int)((int)i = 8))
```

Taking the address of an lvalue cast will not work because the address operator may not be applied to a bit field.

You can also use the following function-style notation to convert the value of *expr* to the type *T*. :

*expr***(** *T* **)**

A function-style cast with no arguments, such as `X()` is equivalent to the declaration `X t()`, where `t` is a temporary object. Similarly, a function-style cast with more than one argument, such as `X(a, b)`, is equivalent to the declaration `X t(a, b)`.

For C++, the operand can have class type. If the operand has class type, it can be cast to any type for which the class has a user-defined conversion function. Casts can invoke a constructor, if the target type is a class, or they can invoke a conversion function, if the source type is a class. They can be ambiguous if both conditions hold.

An explicit type conversion can also be expressed by using the C++ type conversion operator **static_cast**.

Example

The following demonstrates the use of the cast operator. The example dynamically creates an integer array of size 10:

```
#include <stdlib.h>

int main(void) {
   int* myArray = (int*) malloc(10 * sizeof(int));
   free(myArray);
   return 0;
}
```

The `malloc()` library function returns a **void** pointer that points to memory that will hold an object of the size of its argument. The statement `int* myArray = (int*) malloc(10 * sizeof(int))` does the following

- Creates a void pointer that points to memory that can hold ten integers.
- Converts that **void** pointer into an integer pointer with the use of the cast operator.
- Assigns that integer pointer to `myArray`. Because a name of an array is the same as a pointer to the initial element of the array, `myArray` is an array of ten integers stored in the memory created by the call to `malloc()`.

Cast to a Union Type

▶ C Casting to a union type is the ability to cast a union member to the same type as the union to which it belongs. Such a cast does not produce an lvalue, unlike other casts. The feature is supported as an orthogonal extension to C99, implemented to facilitate porting programs developed with GNU C.

Only a type that explicitly exists as a member of a union type can be cast to that union type. The cast can use either the tag of the union type or a union type name declared in a **typedef** expression. The type specified must be a complete union type. An anonymous union type can be used in a cast to a union type, provided that it has a tag or type name. A bit field can be cast to a union type, provided that the union contains a bit field member of the same type, but not necessarily of the same length.

Casting to a nested union is also allowed. In the following example, the **double** type `dd` can be cast to the nested union `u2_t`.

```
int main() {
   union u_t {
      char a;
      short b;
      int c;
      union u2_t {
         double d;
      }u2;
   };
   union u_t U;
   double dd = 1.234;
   U.u2 = (union u2_t) dd;      // Valid.
   printf("U.u2 is %f\n", U.u2);
}
```

The output of this example is:

```
U.u2 is 1.234
```

A union cast is also valid as a function argument, part of a constant expression for initialization, and in a compound literal statement.

Binary Expressions

A *binary expression* contains two operands separated by one operator.

Not all binary operators have the same precedence.

All binary operators have left-to-right associativity.

The order in which the operands of most binary operators are evaluated is not specified. To ensure correct results, avoid creating binary expressions that depend on the order in which the compiler evaluates the operands.

As indicated in the following descriptions, the usual arithmetic conversions are performed on the operands of most binary expressions.

The following table summarizes the operators for binary expressions:

Precedence and associativity of binary operators

Rank	Right Associative?	Operator Function	Usage
5		multiplication	*expr* * *expr*
5		division	*expr* / *expr*
5		modulo (remainder)	*expr* % *expr*
6		binary addition	*expr* + *expr*
6		binary subtraction	*expr* - *expr*
7		bitwise shift left	*expr* << *expr*
7		bitwise shift right	*expr* >> *expr*

Precedence and associativity of binary operators

Rank	Right Associative?	Operator Function	Usage
8		less than	*expr* **<** *expr*
8		less than or equal to	*expr* **<=** *expr*
8		greater than	*expr* **>** *expr*
8		greater than or equal to	*expr* **>=** *expr*
9		equal	*expr* **==** *expr*
9		not equal	*expr* **!=** *expr*
10		bitwise AND	*expr* **&** *expr*
11		bitwise exclusive OR	*expr* **^** *expr*
12		bitwise inclusive OR	*expr* **|** *expr*
13		logical AND	*expr* **&&** *expr*
14		logical inclusive OR	*expr* **||** *expr*
16	yes	simple assignment	*lvalue* **=** *expr*
16	yes	multiply and assign	*lvalue* ***=** *expr*
16	yes	divide and assign	*lvalue* **/=** *expr*
16	yes	modulo and assign	*lvalue* **%=** *expr*
16	yes	add and assign	*lvalue* **+=** *expr*
16	yes	subtract and assign	*lvalue* **-=** *expr*
16	yes	shift left and assign	*lvalue* **<<=** *expr*
16	yes	shift right and assign	*lvalue* **>>=** *expr*
16	yes	bitwise AND and assign	*lvalue* **&=** *expr*
16	yes	bitwise exclusive OR and assign	*lvalue* **^=** *expr*
16	yes	bitwise inclusive OR and assign	*lvalue* **|=** *expr*
18		comma (sequencing)	*expr* **,** *expr*

Related References
- "Operator Precedence and Associativity" on page 99
- "Arithmetic Conversions" on page 156

Multiplication *

The `*` (multiplication) operator yields the product of its operands. The operands must have an arithmetic or enumeration type. The result is not an lvalue. The usual arithmetic conversions on the operands are performed.

Because the multiplication operator has both associative and commutative properties, the compiler can rearrange the operands in an expression that contains more than one multiplication operator. For example, the expression:

```
sites * number * cost
```

can be interpreted in any of the following ways:

```
(sites * number) * cost
sites * (number * cost)
(cost * sites) * number
```

Division /

The `/` (division) operator yields the algebraic quotient of its operands. If both operands are integers, any fractional part (remainder) is discarded. Throwing away the fractional part is often called *truncation toward zero*. The operands must have an arithmetic or enumeration type. The right operand may not be zero: the result is undefined if the right operand evaluates to `0`. For example, expression `7 / 4` yields the value `1` (rather than `1.75` or `2`). The result is not an lvalue.

The usual arithmetic conversions on the operands are performed.

Remainder %

The % (remainder) operator yields the remainder from the division of the left operand by the right operand. For example, the expression 5 % 3 yields 2. The result is not an lvalue.

Both operands must have an integral or enumeration type. If the right operand evaluates to 0, the result is undefined. If either operand has a negative value, the result is such that the following expression always yields the value of a if b is not 0 and a/b is representable:

```
( a / b ) * b + a %b;
```

The usual arithmetic conversions on the operands are performed.

C++ If both operands are negative, the sign of the remainder is also negative. Otherwise, the sign of the remainder is the same as the sign of the quotient.

Addition +

The + (addition) operator yields the sum of its operands. Both operands must have an arithmetic type, or one operand must be a pointer to an object type and the other operand must have an integral or enumeration type.

When both operands have an arithmetic type, the usual arithmetic conversions on the operands are performed. The result has the type produced by the conversions on the operands and is not an lvalue.

A pointer to an object in an array can be added to a value having integral type. The result is a pointer of the same type as the pointer operand. The result refers to another element in the array, offset from the original element by the amount of the integral value treated as a subscript. If the resulting pointer points to storage outside the array, other than the first location outside the array, the result is undefined. A pointer to one element past the end of an array cannot be used to access the memory content at that address. The compiler does not provide boundary checking on the pointers. For example, after the addition, ptr points to the third element of the array:

```
int array[5];
int *ptr;
ptr = array + 2;
```

Subtraction –

The - (subtraction) operator yields the difference of its operands. Both operands must have an arithmetic or enumeration type, or the left operand must have a pointer type and the right operand must have the same pointer type or an integral or enumeration type. You cannot subtract a pointer from an integral value.

When both operands have an arithmetic type, the usual arithmetic conversions on the operands are performed. The result has the type produced by the conversions on the operands and is not an lvalue.

When the left operand is a pointer and the right operand has an integral type, the compiler converts the value of the right to an address offset. The result is a pointer of the same type as the pointer operand.

If both operands are pointers to elements in the same array, the result is the number of objects separating the two addresses. The number is of type **ptrdiff_t**, which is defined in the header file `stddef.h`. Behavior is undefined if the pointers do not refer to objects in the same array.

Bitwise Left and Right Shift << >>

The bitwise shift operators move the bit values of a binary object. The left operand specifies the value to be shifted. The right operand specifies the number of positions that the bits in the value are to be shifted. The result is not an lvalue. Both operands have the same precedence and are left-to-right associative.

Operator	Usage
<<	Indicates the bits are to be shifted to the left.
>>	Indicates the bits are to be shifted to the right.

Each operand must have an integral or enumeration type. The compiler performs integral promotions on the operands, and then the right operand is converted to type **int**. The result has the same type as the left operand (after the arithmetic conversions).

The right operand should not have a negative value or a value that is greater than or equal to the width in bits of the expression being shifted. The result of bitwise shifts on such values is unpredictable.

If the right operand has the value `0`, the result is the value of the left operand (after the usual arithmetic conversions).

The << operator fills vacated bits with zeros. For example, if `left_op` has the value `4019`, the bit pattern (in 16-bit format) of `left_op` is:

```
0000111110110011
```

The expression `left_op << 3` yields:

```
0111110110011000
```

The expression `left_op >> 3` yields:

```
0000000111110110
```

Relational < > <= >=

The relational operators compare two operands and determine the validity of a relationship.

C The type of the result is **int** and has the values `1` if the specified relationship is true, and `0` if false.

C++ The type of the result is **bool** and has the values **true** or **false**.

The result is not an lvalue.

The following table describes the four relational operators:

Operator	Usage
<	Indicates whether the value of the left operand is less than the value of the right operand.
>	Indicates whether the value of the left operand is greater than the value of the right operand.
<=	Indicates whether the value of the left operand is less than or equal to the value of the right operand.
>=	Indicates whether the value of the left operand is greater than or equal to the value of the right operand.

Both operands must have arithmetic or enumeration types or be pointers to the same type.

▶ C The result has type **int**.

▶ C++ The result has type **bool**.

If the operands have arithmetic types, the usual arithmetic conversions on the operands are performed.

When the operands are pointers, the result is determined by the locations of the objects to which the pointers refer. If the pointers do not refer to objects in the same array, the result is not defined.

A pointer can be compared to a constant expression that evaluates to 0. You can also compare a pointer to a pointer of type **void***. The pointer is converted to a pointer of type **void***.

If two pointers refer to the same object, they are considered equal. If two pointers refer to nonstatic members of the same object, the pointer to the object declared later is greater, provided that they are not separated by an access specifier; otherwise the comparison is undefined. If two pointers refer to data members of the same union, they have the same address value.

If two pointers refer to elements of the same array, or to the first element beyond the last element of an array, the pointer to the element with the higher subscript value is greater.

You can only compare members of the same object with relational operators.

Relational operators have left-to-right associativity. For example, the expression:

```
a < b <= c
```

is interpreted as:

```
(a < b) <= c
```

If the value of `a` is less than the value of `b`, the first relationship yields 1 in C, or **true** in C++. The compiler then compares the value **true** (or 1) with the value of `c` (integral promotions are carried out if needed).

Equality == !=

The equality operators, like the relational operators, compare two operands for the validity of a relationship. The equality operators, however, have a lower precedence than the relational operators.

C The type of the result is **int** and has the values 1 if the specified relationship is true, and 0 if false.

C++ The type of the result is **bool** and has the values **true** or **false**.

The following table describes the two equality operators:

Operator	Usage
==	Indicates whether the value of the left operand is equal to the value of the right operand.
!=	Indicates whether the value of the left operand is not equal to the value of the right operand.

Both operands must have arithmetic or enumeration types or be pointers to the same type, or one operand must have a pointer type and the other operand must be a pointer to void or a null pointer. The result is type **int** in C or **bool** in C++.

If the operands have arithmetic types, the usual arithmetic conversions on the operands are performed.

If the operands are pointers, the result is determined by the locations of the objects to which the pointers refer.

If one operand is a pointer and the other operand is an integer having the value 0, the == expression is true only if the pointer operand evaluates to NULL. The != operator evaluates to true if the pointer operand does *not* evaluate to NULL.

You can also use the equality operators to compare pointers to members that are of the same type but do not belong to the same object. The following expressions contain examples of equality and relational operators:

```
time < max_time == status < complete
letter != EOF
```

Note: The equality operator (==) should not be confused with the assignment (=) operator.

For example,

`if (x == 3)` evaluates to **true** (or 1) if x is equal to three. Equality tests like this should be coded with spaces between the operator and the operands to prevent unintentional assignments.

while

`if (x = 3)` is taken to be true because (x = 3) evaluates to a nonzero value (3). The expression also assigns the value 3 to x.

Related References

- “Simple Assignment =” on page 145

Bitwise AND &

The & (bitwise AND) operator compares each bit of its first operand to the corresponding bit of the second operand. If both bits are 1's, the corresponding bit of the result is set to 1. Otherwise, it sets the corresponding result bit to 0.

Both operands must have an integral or enumeration type. The usual arithmetic conversions on each operand are performed. The result has the same type as the converted operands.

Because the bitwise AND operator has both associative and commutative properties, the compiler can rearrange the operands in an expression that contains more than one bitwise AND operator.

The following example shows the values of a, b, and the result of a & b represented as 16-bit binary numbers:

bit pattern of a	0000000001011100
bit pattern of b	0000000000101110
bit pattern of a & b	0000000000001100

Note: The bitwise AND (&) should not be confused with the logical AND. (&&) operator. For example,

```
   1 & 4 evaluates to 0
while
   1 && 4 evaluates to true
```

Related References

- "Logical AND &&" on page 140

Bitwise Exclusive OR ^

The bitwise exclusive OR operator (in EBCDIC, the ^ symbol is represented by the ¬ symbol) compares each bit of its first operand to the corresponding bit of the second operand. If both bits are 1's or both bits are 0's, the corresponding bit of the result is set to 0. Otherwise, it sets the corresponding result bit to 1.

Both operands must have an integral or enumeration type. The usual arithmetic conversions on each operand are performed. The result has the same type as the converted operands and is not an lvalue.

Because the bitwise exclusive OR operator has both associative and commutative properties, the compiler can rearrange the operands in an expression that contains more than one bitwise exclusive OR operator. Note that the ^ character can be represented by the trigraph ??'.

The following example shows the values of a, b, and the result of a ^ b represented as 16-bit binary numbers:

bit pattern of a	0000000001011100
bit pattern of b	0000000000101110
bit pattern of a ^ b	0000000001110010

Related References

- "Trigraph Sequences" on page 15

Bitwise Inclusive OR |

The | (bitwise inclusive OR) operator compares the values (in binary format) of each operand and yields a value whose bit pattern shows which bits in either of the operands has the value 1. If both of the bits are 0, the result of that bit is 0; otherwise, the result is 1.

Both operands must have an integral or enumeration type. The usual arithmetic conversions on each operand are performed. The result has the same type as the converted operands and is not an lvalue.

Because the bitwise inclusive OR operator has both associative and commutative properties, the compiler can rearrange the operands in an expression that contains more than one bitwise inclusive OR operator. Note that the | character can be represented by the trigraph ??!.

The following example shows the values of a, b, and the result of a | b represented as 16-bit binary numbers:

bit pattern of a	0000000001011100
bit pattern of b	0000000000101110
bit pattern of a | b	0000000001111110

Note: The bitwise OR (|) should not be confused with the logical OR (||) operator. For example,

```
   1 | 4 evaluates to 5
while
   1 || 4 evaluates to true
```

Related References
- "Trigraph Sequences" on page 15
- "Logical OR ||" on page 141

Logical AND &&

The && (logical AND) operator indicates whether both operands are true.

C If both operands have nonzero values, the result has the value 1. Otherwise, the result has the value 0. The type of the result is **int**. Both operands must have a arithmetic or pointer type. The usual arithmetic conversions on each operand are performed.

C++ If both operands have values of **true**, the result has the value **true**. Otherwise, the result has the value **false**. Both operands are implicitly converted to **bool** and the result type is **bool**.

Unlike the & (bitwise AND) operator, the && operator guarantees left-to-right evaluation of the operands. If the left operand evaluates to 0 (or **false**), the right operand is not evaluated.

The following examples show how the expressions that contain the logical AND operator are evaluated:

Expression	Result
`1 && 0`	`false` or `0`
`1 && 4`	`true` or `1`
`0 && 0`	`false` or `0`

The following example uses the logical AND operator to avoid division by zero:

```
(y != 0) && (x / y)
```

The expression `x / y` is not evaluated when `y != 0` evaluates to `0` (or **false**).

Note: The logical AND (`&&`) should not be confused with the bitwise AND (`&`) operator. For example:

`1 && 4` evaluates to `1` (or **true**)
while
`1 & 4` evaluates to `0`

Related References
- "Bitwise AND &" on page 139

Logical OR ||

The `||` (logical OR) operator indicates whether either operand is true.

C If either of the operands has a nonzero value, the result has the value `1`. Otherwise, the result has the value `0`. The type of the result is **int**. Both operands must have a arithmetic or pointer type. The usual arithmetic conversions on each operand are performed.

C++ If either operand has a value of **true**, the result has the value **true**. Otherwise, the result has the value **false**. Both operands are implicitly converted to **bool** and the result type is **bool**.

Unlike the `|` (bitwise inclusive OR) operator, the `||` operator guarantees left-to-right evaluation of the operands. If the left operand has a nonzero (or **true**) value, the right operand is not evaluated.

The following examples show how expressions that contain the logical OR operator are evaluated:

Expression	Result
`1 || 0`	`true` or `1`
`1 || 4`	`true` or `1`
`0 || 0`	`false` or `0`

The following example uses the logical OR operator to conditionally increment `y`:

```
++x || ++y;
```

The expression `++y` is not evaluated when the expression `++x` evaluates to a nonzero (or **true**) quantity.

Note: The logical OR (`||`) should not be confused with the bitwise OR (`|`) operator. For example:

1 || 4 evaluates to 1 (or **true**)
while
1 | 4 evaluates to 5

Related References

- "Bitwise Inclusive OR |" on page 140

C++ Pointer to Member Operators .* –>*

C++ There are two pointer to member operators: `.*` and `->*`.

The `.*` operator is used to dereference pointers to class members. The first operand must be of class type. If the type of the first operand is class type `T`, or is a class that has been derived from class type `T`, the second operand must be a pointer to a member of a class type `T`.

The `->*` operator is also used to dereference pointers to class members. The first operand must be a pointer to a class type. If the type of the first operand is a pointer to class type `T`, or is a pointer to a class derived from class type `T`, the second operand must be a pointer to a member of class type `T`.

The `.*` and `->*` operators bind the second operand to the first, resulting in an object or function of the type specified by the second operand.

If the result of`.*` or `->*` is a function, you can only use the result as the operand for the `( )` (function call) operator. If the second operand is an lvalue, the result of `.*` or `->*` is an lvalue.

Related References

- "Lvalues and Rvalues" on page 103
- "Pointers to Members" on page 268

Conditional Expressions

A *conditional expression* is a compound expression that contains a condition that is implicitly converted to type `bool` in C++(*operand*$_1$), an expression to be evaluated if the condition evaluates to true (*operand*$_2$), and an expression to be evaluated if the condition has the value false (*operand*$_3$).

The conditional expression contains one two-part operator. The `?` symbol follows the condition, and the `:` symbol appears between the two action expressions. All expressions that occur between the `?` and `:` are treated as one expression.

The first operand must have a scalar type. The type of the second and third operands must be one of the following:

- An arithmetic type
- A compatible pointer, structure, or union type
- void

The second and third operands can also be a pointer or a null pointer constant.

Two objects are compatible when they have the same type but not necessarily the same type qualifiers (**volatile** or **const**). Pointer objects are compatible if they have the same type or are pointers to void.

The first operand is evaluated, and its value determines whether the second or third operand is evaluated:

- If the value is true, the second operand is evaluated.
- If the value is false, the third operand is evaluated.

The result is the value of the second or third operand.

If the second and third expressions evaluate to arithmetic types, the usual arithmetic conversions are performed on the values. The types of the second and third operands determine the type of the result as shown in the following tables.

Conditional expressions have right-to-left associativity with respect to their first and third operands. The leftmost operand is evaluated first, and then only one of the remaining two operands is evaluated. The following expressions are equivalent:

```
a ? b : c ? d : e ? f : g
a ? b : (c ? d : (e ? f : g))
```

Type of Conditional C Expressions

In C, a conditional expression is not an lvalue, nor is its result.

Type of One Operand	Type of Other Operand	Type of Result
Arithmetic	Arithmetic	Arithmetic type after usual arithmetic conversions
Structure or union type	Compatible structure or union type	Structure or union type with all the qualifiers on both operands
void	**void**	**void**
Pointer to compatible type	Pointer to compatible type	Pointer to type with all the qualifiers specified for the type
Pointer to type	`NULL` pointer (the constant `0`)	Pointer to type
Pointer to object or incomplete type	Pointer to **void**	Pointer to **void** with all the qualifiers specified for the type

▶ C In GNU C, a conditional expression is a valid lvalue, provided that its type is not **void** and both of its branches are valid lvalues. The following conditional expression `(a ? b : c)` is legal under GNU C:

```
(a ? b : c) = 5
/*  Under GNU C, equivalent to (a ? b = 5 : (c = 5))  */
```

This extension is available when compiling in one of the extended language levels.

Type of Conditional C++ Expressions

In C++, a conditional expression is a valid lvalue if its type is not **void**, and its result is an lvalue.

Type of One Operand	Type of Other Operand	Type of Result
Reference to type	Reference to type	Reference after usual reference conversions
Class `T`	Class `T`	Class `T`

Type of One Operand	Type of Other Operand	Type of Result
Class T	Class X	Class type for which a conversion exists. If more than one possible conversion exists, the result is ambiguous.
throw expression	Other (type, pointer, reference)	Type of the expression that is not a **throw** expression

Examples of Conditional Expressions

The following expression determines which variable has the greater value, y or z, and assigns the greater value to the variable x:

```
x = (y > z) ? y : z;
```

The following is an equivalent statement:

```
if (y > z)
   x = y;
else
   x = z;
```

The following expression calls the function printf, which receives the value of the variable c, if c evaluates to a digit. Otherwise, printf receives the character constant 'x'.

```
printf(" c = %c\n", isdigit(c) ? c : 'x');
```

If the last operand of a conditional expression contains an assignment operator, use parentheses to ensure the expression evaluates properly. For example, the = operator has higher precedence than the ?: operator in the following expression:

```
int i,j,k;
(i == 7) ? j ++ : k = j;
```

The compiler will interpret this expression as if it were parenthesized this way:

```
int i,j,k;
((i == 7) ? j ++ : k) = j;
```

That is, k is treated as the third operand, not the entire assignment expression k = j.

To assign the value of j to k when i == 7 is false, enclose the last operand in parentheses:

```
int i,j,k;
(i == 7) ? j ++ : (k = j);
```

Assignment Expressions

An *assignment expression* stores a value in the object designated by the left operand. There are two types of assignment operators: simple assignment and compound assignment.

The left operand in all assignment expressions must be a modifiable lvalue. The type of the expression is the type of the left operand. The value of the expression is the value of the left operand after the assignment has completed.

The result of an assignment expression is not an lvalue in C, but is an lvalue in C++.

All assignment operators have the same precedence and have right-to-left associativity.

Simple Assignment =

The simple assignment operator has the following form:

lvalue = *expr*

The operator stores the value of the right operand *expr* in the object designated by the left operand *lvalue*.

The left operand must be a modifiable lvalue. The type of an assignment operation is the type of the left operand.

If the left operand is not a class type or a vector type, the right operand is implicitly converted to the type of the left operand. This converted type will not be qualified by **const** or **volatile**.

If the left operand is a class type, that type must be complete. The copy assignment operator of the left operand will be called.

If the left operand is an object of reference type, the compiler will assign the value of the right operand to the object denoted by the reference.

Linux Mac OS X The assignment operator has been extended to permit operands of vector type. Both sides of an assignment expression must be of the same vector type.

Related References
- "Lvalues and Rvalues" on page 103
- "Pointers" on page 83
- "Type Qualifiers" on page 71

Compound Assignment

The compound assignment operators consist of a binary operator and the simple assignment operator. They perform the operation of the binary operator on both operands and store the result of that operation into the left operand, which must be a modifiable lvalue.

The following table shows the operand types of compound assignment expressions:

Operator	Left Operand	Right Operand
+= or -=	Arithmetic	Arithmetic
+= or -=	Pointer	Integral type
*=, /=, and %=	Arithmetic	Arithmetic
<<=, >>=, &=, ^=, and |=	Integral type	Integral type

Note that the expression

```
a *= b + c
```

is equivalent to

```
a = a * (b + c)
```

and *not*

```
a = a * b + c
```

The following table lists the compound assignment operators and shows an expression using each operator:

Operator	Example	Equivalent Expression
+=	index += 2	index = index + 2
-=	*(pointer++) -= 1	*pointer = *(pointer++) - 1
*=	bonus *= increase	bonus = bonus * increase
/=	time /= hours	time = time / hours
%=	allowance %= 1000	allowance = allowance % 1000
<<=	result <<= num	result = result << num
>>=	form >>= 1	form = form >> 1
&=	mask &= 2	mask = mask & 2
^=	test ^= pre_test	test = test ^ pre_test
|=	flag |= ON	flag = flag | ON

Although the equivalent expression column shows the left operands (from the example column) twice, it is in effect evaluated only once.

C++ In addition to the table of operand types, an expression is implicitly converted to the cv-unqualified type of the left operand if it is not of class type. However, if the left operand is of class type, the class becomes complete, and assignment to objects of the class behaves as a copy assignment operation. Compound expressions and conditional expressions are lvalues in C++, which allows them to be a left operand in a compound assignment expression.

C When GNU C language features have been enabled, compound expressions and conditional expressions are allowed as lvalues, provided that their operands are lvalues. The following compound assignment of the compound expression (a, b) is legal under GNU C, provided that expression b, or more generally, the last expression in the sequence, is an lvalue:

```
(a,b) += 5  /* Under GNU C, this is equivalent to
a, (b += 5)    */
```

Comma Expressions

A *comma expression* contains two operands of any type separated by a comma and has left-to-right associativity. The left operand is fully evaluated, possibly producing side effects, and its value, if there is one, is discarded. The right operand is then evaluated. The type and value of the result of a comma expression are those of its right operand, after the usual unary conversions. In C, the result of a comma expression is not an lvalue. In C++, the result is an lvalue if the right operand is an lvalue. The following statements are equivalent:

```
r = (a,b,...,c);
a; b; r = c;
```

The difference is that the comma operator may be suitable for expression contexts, such as loop control expressions.

Similarly, the address of a compound expression can be taken if the right operand is an lvalue.

```
&(a, b)
a, &b
```

Any number of expressions separated by commas can form a single expression because the comma operator is associative. The use of the comma operator guarantees that the subexpressions will be evaluated in left-to-right order, and the value of the last becomes the value of the entire expression.

In the following example, if omega has the value 11, the expression increments delta and assigns the value 3 to alpha:

```
alpha = (delta++, omega % 4);
```

A sequence point occurs after the evaluation of the first operand. The value of delta is discarded.

For example, the value of the expression:

```
intensity++, shade * increment, rotate(direction);
```

is the value of the expression:

```
rotate(direction)
```

The primary use of the comma operator is to produce side effects in the following situations:

- Calling a function
- Entering or repeating an iteration loop
- Testing a condition
- Other situations where a side effect is required but the result of the expression is not immediately needed

In some contexts where the comma character is used, parentheses are required to avoid ambiguity. For example, the function

```
f(a, (t = 3, t + 2), c);
```

has only three arguments: the value of a, the value 5, and the value of c. Other contexts in which parentheses are required are in field-length expressions in structure and union declarator lists, enumeration value expressions in enumeration declarator lists, and initialization expressions in declarations and initializers.

In the previous example, the comma is used to separate the argument expressions in a function invocation. In this context, its use does not guarantee the order of evaluation (left to right) of the function arguments.

The following table gives some examples of the uses of the comma operator:

Statement	Effects
for (i=0; i<2; ++i, f());	A **for** statement in which i is incremented and f() is called at each iteration.

Statement	Effects

`if ( f(), ++i, i>1 )`
`   { /* ... */ }`
	An **if** statement in which function `f()` is called, variable `i` is incremented, and variable `i` is tested against a value. The first two expressions within this comma expression are evaluated before the expression `i>1`. Regardless of the results of the first two expressions, the third is evaluated and its result determines whether the **if** statement is processed.

`func( ( ++a, f(a) ) );`	A function call to `func()` in which `a` is incremented, the resulting value is passed to a function `f()`, and the return value of `f()` is passed to `func()`. The function `func()` is passed only a single argument, because the comma expression is enclosed in parentheses within the function argument list.

C++ throw Expressions

▶ C++ A *throw* expression is used to throw exceptions to C++ exception handlers. A throw expression is of type **void**.

Related References

- Chapter 17, “Exception Handling,” on page 369
- “void Type” on page 53

Chapter 6. Implicit Type Conversions

An expression `e` of a given type is *implicitly converted* if used in one of the following situations:

- Expression `e` is used as an operand of an arithmetic or logical operation.
- Expression `e` is used as a condition in an **if** statement or an iteration statement (such as a **for** loop). Expression `e` will be converted to **bool** (or **int** in C).
- Expression `e` is used in a **switch** statement. Expression `e` will be converted to an integral type.
- Expression `e` is used in an initialization. This includes the following:
  - An assignment is made to an lvalue that has a different type than `e`.
  - A function is provided an argument value of `e` that has a different type than the parameter.
  - Expression `e` is specified in the **return** statement of a function, and `e` has a different type from the defined return type for the function.

The compiler will allow an implicit conversion of an expression `e` to a type `T` if and only if the compiler would allow the following statement:

```
T var = e;
```

For example when you add values having different data types, both values are first converted to the same type: when a **short int** value and an **int** value are added together, the **short int** value is converted to the **int** type.

You can perform explicit type conversions using one of the cast operators, the function style cast, or the C-style cast.

Integral and Floating-Point Promotions

An *integral promotion* is the conversion of one integral type to another where the second type can hold all possible values of the first type. Certain fundamental types can be used wherever an integer can be used. The following fundamental types can be converted through integral promotion are:

- **char**
- C++ **bool**
- **wchar_t**
- **short int**
- enumerators
- objects of enumeration type
- integer bit fields (both signed and unsigned)

Except for **wchar_t**, if the value cannot be represented by an **int**, the value is converted to an **unsigned int**. For **wchar_t**, if an **int** can represent all the values of the original type, the value is converted to the type that can best represent all the values of the original type. For example, if a **long** can represent all the values, the value is converted to a **long**.

Floating-Point Promotions

You can convert an rvalue of type **float** to an rvalue of type **double**. The value of the expression is unchanged. This conversion is a *floating-point promotion*.

Standard Type Conversions

Many C and C++ operators cause *implicit type conversions*, which change the type of an expression. When you add values having different data types, both values are first converted to the same type. For example, when a **short int** value and an **int** value are added together, the **short int** value is converted to the **int** type. It can result in loss of data if the value of the original object is outside the range representable by the shorter type.

Implicit type conversions can occur when:
- An operand is prepared for an arithmetic or logical operation.
- An assignment is made to an lvalue that has a different type than the assigned value.
- A function is provided an argument value that has a different type than the parameter.
- The value specified in the **return** statement of a function has a different type from the defined return type for the function.

You can perform explicit type conversions using the C-style cast, the C++ function-style cast, or one of the C++ cast operators.

```
#include <iostream>
using namespace std;

int main() {
  float num = 98.76;
  int x1 = (int) num;
  int x2 = int(num);
  int x3 = static_cast<int>(num);

  cout << "x1 = " << x1 << endl;
  cout << "x2 = " << x2 << endl;
  cout << "x3 = " << x3 << endl;
}
```

The following is the output of the above example:

```
x1 = 98
x2 = 98
x3 = 98
```

The integer `x1` is assigned a value in which `num` has been explicitly converted to an **int** with the C-style cast. The integer `x2` is assigned a value that has been converted with the function-style cast. The integer `x3` is assigned a value that has been converted with the **static_cast** operator.

Related References
- "User-Defined Conversions" on page 328

Lvalue-to-Rvalue Conversions

If an lvalue appears in a situation in which the compiler expects an rvalue, the compiler converts the lvalue to an rvalue.

An lvalue `e` of a type `T` can be converted to an rvalue if `T` is not a function or array type. The type of `e` after conversion will be `T`. The following table lists exceptions to this:

Situation before conversion	Resulting behavior
`T` is an incomplete type	compile-time error

Situation before conversion	Resulting behavior
e refers to an uninitialized object	undefined behavior
e refers to an object not of type T	undefined behavior
C++ e refers to an object not of type T, nor a type derived from T	undefined behavior
C++ T is a nonclass type	the type of e after conversion is T, not qualified by either **const** or **volatile**

Related References
- "Lvalues and Rvalues" on page 103

Boolean Conversions

C The conversion of any scalar value to type **_Bool** has a result of 0 if the value compares equal to 0; otherwise the result is 1.

C++ You can convert integral, floating-point, arithmetic, enumeration, pointer, and pointer to member rvalue types to an rvalue of type **bool**. A zero, null pointer, or null member pointer value is converted to **false**. All other values are converted to **true**.

The following is an example of boolean conversions:

```
void f(int* a, int b)
{
  bool d = a;  // false if a == NULL
  bool e = b;  // false if b == 0
}
```

The variable d will be **false** if a is equal to a null pointer. Otherwise, d will be **true**. The variable e will be **false** if b is equal to zero. Otherwise, e will be **true**.

Related References
- "Boolean Variables" on page 49

Integral Conversions

You can convert the following:
- An rvalue of integer type (including signed and unsigned integer types) to another rvalue of integer type
- An rvalue of enumeration type to an rvalue of integer type

If you are converting an integer a to an unsigned type, the resulting value x is the least unsigned integer such that a and x are congruent modulo 2^*n*, where n is the number of bits used to represent an unsigned type. If two numbers a and x are congruent modulo 2^*n*, the following expression is true, where the function pow(m, n) returns the value of m to the power of n:

```
a % pow(2, n) == x % pow(2, n)
```

If you are converting an integer a to a signed type, the compiler does not change the resulting value if the new type is large enough to hold a. If the new type is not large enough, the behavior is defined by the compiler.

C++ If you are converting a **bool** to an integer, values of **false** are converted to 0; values of **true** are converted to 1.

Integer promotions belong to a different category of conversions; they are not integral conversions.

Related References

- "Integer Variables" on page 52

Floating-Point Conversions

You can convert an rvalue of floating-point type to another rvalue of floating-point type.

Floating-point promotions (converting from **float** to **double**) belong to a different category of conversions; they are not floating-point conversions.

Related References

- "Floating-Point Variables" on page 51
- "Integral and Floating-Point Promotions" on page 149

Pointer Conversions

Pointer conversions are performed when pointers are used, including pointer assignment, initialization, and comparison.

▶ C Conversions that involve pointers must use an explicit type cast. The exceptions to this rule are the allowable assignment conversions for C pointers. In the following table, a `const`-qualified lvalue cannot be used as a left operand of the assignment.

Table 1. Legal assignment conversions for C pointers

Left operand type	Permitted right operand types
pointer to (object) `T`	the constant `0` a pointer to a type compatible with `T` a pointer to `void` (**void***)
pointer to (function) `F`	the constant `0` a pointer to a function compatible with `F`

The referenced type of the left operand must have the same qualifiers as the right operand. An object pointer may be an incomplete type if the other pointer has type **void***.

C pointers are not necessarily the same size as type **int**. Pointer arguments given to functions should be explicitly cast to ensure that the correct type expected by the function is being passed. The generic object pointer in C is **void***, but there is no generic function pointer.

Conversion to void*

Any pointer to an object of a type `T`, optionally type-qualified, can be converted to **void***, keeping the same **const** or **volatile** qualifications.

▶ C The allowable assignment conversions involving **void*** as the left operand are shown in the following table.

*Table 2. Legal assignment conversions in C for void**

Left operand type	Permitted right operand types
(**void***)	the constant `0` a pointer to (object) `T` (**void***)

The object `T` may be an incomplete type.

C++ Pointers to functions cannot be converted to the type **void*** with a standard conversion: this can be accomplished explicitly, provided that a **void*** has sufficient bits to hold it.

Derived-to-Base Conversions

C++ You can convert an rvalue pointer of type `B*` to an rvalue pointer of class `A*` where `A` is an accessible base class of `B` as long as the conversion is not ambiguous. The conversion is ambiguous if the expression for the accessible base class can refer to more than one distinct class. The resulting value points to the base class subobject of the derived class object. If the pointer of type `B*` is null, it will be converted to a null pointer of type `A*`. Note that you cannot convert a pointer to a class into a pointer to its base class if the base class is a virtual base class of the derived class.

Null Pointer Constants

A constant expression that evaluates to zero is a *null pointer constant*. This expression can be converted to a pointer. This pointer will be a null pointer (pointer with a zero value), and is guaranteed not to point to any object.

Array-to-Pointer Conversions

You can convert an lvalue or rvalue with type "array of `N`," where `N` is the type of a single element of the array, to `N*`. The result is a pointer to the initial element of the array. You cannot perform the conversion if the expression is used as the operand of the `&` (address) operator or the **sizeof** operator.

Function-to-Pointer Conversions

You can convert an lvalue that is a function of type `T` to an rvalue that is a pointer to a function of type `T`, except when the expression is used as the operand of the `&` (address) operator, the `()` (function call) operator, or the **sizeof** operator.

Reference Conversions

C++ A reference conversion can be performed wherever a reference initialization occurs, including reference initialization done in argument passing and function return values. A reference to a class can be converted to a reference to an accessible base class of that class as long as the conversion is not ambiguous. The result of the conversion is a reference to the base class subobject of the derived class object.

Reference conversion is allowed if the corresponding pointer conversion is allowed.

Pointer-to-Member Conversions

C++ Pointer-to-member conversion can occur when pointers to members are initialized, assigned, or compared. Note that pointer to a member is not the same as a pointer to an object or a pointer to a function.

A constant expression that evaluates to zero can be converted to the null pointer to a member.

A pointer to a member of a base class can be converted to a pointer to a member of a derived class if the following conditions are true:
- The conversion is not ambiguous. The conversion is ambiguous if multiple instances of the base class are in the derived class.
- A pointer to the derived class can be converted to a pointer to the base class. If this is the case, the base class is said to be *accessible*.
- Member types must match. For example suppose class `A` is a base class of class `B`. You cannot convert a pointer to member of `A` of type **int** to a pointer to member of type `B` of type **float**.
- The base class cannot be virtual.

Qualification Conversions

C++ You can convert an rvalue of type *cv1* `T*` where *cv1* is any combination of zero or more **const** or **volatile** qualifications, to an rvalue of type *cv2* `T*` if *cv2* `T*` is more **const** or **volatile** qualified than *cv1* `T*`.

You can convert an rvalue of type pointer to member of a class `X` of *cv1* `T`, to an rvalue of type pointer to member of a class `X` of *cv2* `T` if *cv2* `T` is more **const** or **volatile** qualified than *cv1* `T`.

Related References
- "Type Qualifiers" on page 71

Function Argument Conversions

C If a function declaration is present and includes declared argument types, the compiler performs type checking. If no function declaration is visible when a function is called, or when an expression appears as an argument in the variable part of a prototype argument list, the compiler performs default argument promotions or converts the value of the expression before passing any arguments to the function. The automatic conversions consist of the following:
- Integral promotions
- Arguments with type **float** are converted to type **double**.

C When compiled using a compiler option that allows the GNU C semantics, a function prototype may override a later K&R nonprototype definition. This behavior is illegal in ISO C. Under ISO C, the type of function arguments after automatic conversion must match that of the function prototype.

```
int func(char);     /* Legal in GCC, illegal in ISO C                      */

int func(ch)        /* ch is automatically promoted to int,                */
   char ch;         /* which does not match the prototype argument type char  */
{ return ch == 0;}


int func(float);     /* Legal in GCC, illegal in ISO C                      */
int func(ch)         /* ch is automatically promoted to double,             */
   float ch;         /* which does not match the prototype argument type float */
{ return ch == 0;}
```

C++ Function declarations in C++ must always specify their parameter types. Also, functions may not be called if it has not already been declared.

Related References
- “Integral and Floating-Point Promotions” on page 149
- “Function Declarations” on page 160

Other Conversions

The void type

By definition, the **void** type has no value. Therefore, it cannot be converted to any other type, and no other value can be converted to **void** by assignment. However, a value can be explicitly cast to **void**.

Structure or union types

No conversions between structure or union types are allowed, except for the following. In C, an assignment conversion between compatible structure or union types is allowed if the right operand is of a type compatible with that of the left operand.

Table 3. Legal assignment conversions in C for structure or union types

Left operand type	**Permitted right operand types**
C a structure or union type	a compatible structure or union type

Class types

C++ There are no standard conversions between class types, but you can write your own conversion operators for class types.

Enumeration types

C In C, when you define a value using the **enum** type specifier, the value is treated as an **int**. Conversions to and from an **enum** value proceed as for the **int** type.

You can convert from an **enum** to any integral type but not from an integral type to an **enum**.

Related References
- “void Type” on page 53
- “User-Defined Conversions” on page 328
- “Enumerations” on page 67

Arithmetic Conversions

The conversions depend on the specific operator and the type of the operand or operands. However, many operators perform similar conversions on operands of integer and floating-point types. These standard conversions are known as the *arithmetic conversions* because they apply to the types of values ordinarily used in arithmetic.

Arithmetic conversions are used for matching operands of arithmetic operators.

Arithmetic conversion proceeds in the following order:

Operand Type	Conversion
One operand has **long double** type	The other operand is converted to **long double**.
One operand has **double** type	The other operand is converted to **double**.
One operand has **float** type	The other operand is converted to **float**.
One operand has **unsigned long long int** type	The other operand is converted to **unsigned long long int**
One operand has **long long** type.	The other operand is converted to **long long**.
One operand has **unsigned long int** type	The other operand is converted to **unsigned long int**.
One operand has **unsigned int** type and the other operand has **long int** type and the value of the **unsigned int** can be represented in a **long int**	The operand with **unsigned int** type is converted to **long int**.
One operand has **unsigned int** type and the other operand has **long int** type and the value of the **unsigned int** cannot be represented in a **long int**	Both operands are converted to **unsigned long int**.
One operand has **long int** type	The other operand is converted to **long int**.
One operand has **unsigned int** type	The other operand is converted to **unsigned int**.
Both operands have **int** type	The result is type **int**.

Related References
- Chapter 5, “Expressions and Operators,” on page 99
- “Integer Variables” on page 52
- “Floating-Point Variables” on page 51

The explicit Keyword

C++ A constructor declared with only one argument and without the `explicit` keyword is a *converting constructor*. You can construct objects with a converting constructor using the assignment operator. Declaring a constructor of this type with the `explicit` keyword prevents this behavior. The *explicit* keyword controls unwanted implicit type conversions. It can only be used in declarations of constructors within a class declaration. For example, except for the default constructor, the constructors in the following class are converting constructors.

```
class A
{  public:
   A();
   A(int);
   A(const char*, int = 0);
};
```

The following declarations are legal.

```
A c = 1;
A d = "Venditti";
```

The first declaration is equivalent to `A c = A(1)`.

If you declare the constructor of the class with the `explicit` keyword, the previous declarations would be illegal.

For example, if you declare the class as:

```
class A
{  public:
   explicit A();
   explicit A(int);
   explicit A(const char*, int = 0);
};
```

You can only assign values that match the values of the class type.

For example, the following statements will be legal:

```
  A a1;
  A a2 = A(1);
  A a3(1);
  A a4 = A("Venditti");
  A* p = new A(1);
  A a5 = (A)1;
  A a6 = static_cast<A>(1);
```

Related References
- "Conversion by Constructor" on page 330

Chapter 7. Functions

In the context of programming languages, the term *function* means an assemblage of statements used for computing an output value. The word is used less strictly than in mathematics, where it means a set relating input variables *uniquely* to output variables. Functions in C or C++ programs may not produce consistent outputs for all inputs, may not produce output at all, or may have side effects. Functions can be understood as user-defined operations, in which the parameters of the parameter list, if any, are the operands.

Functions fall into two categories: those written by you and those provided with the C language implementation. The latter are called *library functions*, since they belong to the library of functions supplied with the compiler.

The result of a function is called its *return value*. The data type of the return value is called the *return type*. A function identifier preceded by its return type and followed by its parameter list is called a *function declaration* or *function prototype*. The term *function body* refers to the statements that represent the actions that the function performs. The body of a function is enclosed in braces, which creates what is called a *function block*. The function return type, followed by its name, parameter list, and body constitute the *function definition*.

The function name followed by the function call operator, (), causes evaluation of the function. If the function has been defined to receive parameters, the values that are to be sent into the function are listed inside the parentheses of the function call operator. These values are the *arguments* for the parameters, and the process just described is called *passing arguments* to the function.

C++ In C++, the parameter list of a function is referred to as its *signature*. The name and signature of a function uniquely identify it. As the word itself suggests, the function signature is used by the compiler to distinguish among the different instances of overloaded functions.

Related References
- Chapter 11, “Overloading,” on page 237

C++ Enhancements to C Functions

C++ The C++ language provides many enhancements to C functions. These are:
- Reference arguments
- Default arguments
- Reference return types
- Member functions
- Overloaded functions
- Operator functions
- Constructor and destructor functions
- Conversion functions
- Virtual functions
- Function templates
- Exception specifications
- Constructor initializers

Related References
- "Passing Arguments by Reference" on page 179
- "Default Arguments in C++ Functions" on page 181
- "Using References as Return Types" on page 184
- "Member Functions" on page 264
- "Overloading Functions" on page 237
- "Overloading Operators" on page 239
- "Constructors and Destructors Overview" on page 311
- "Conversion Functions" on page 331
- "Virtual Functions" on page 302

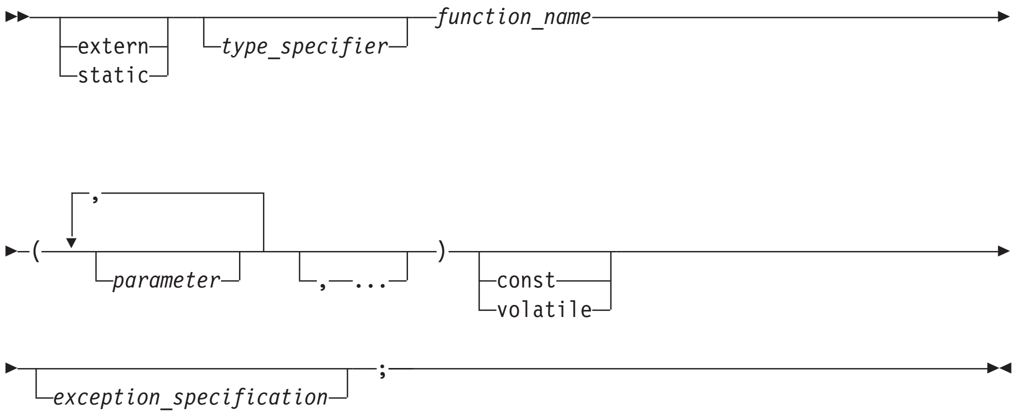
Function Declarations

A function *declaration* establishes the name of the function and the number and types of its parameters. A function declaration consists of a return type, a name, and a parameter list. In addition, a function declaration may optionally specify the function's linkage. In C++, the declaration can also specify an exception specification, a const-qualification, or a volatile-qualification.

A declaration informs the compiler of the format and existence of a function prior to its use. A function can be declared several times in a program, provided that all the declarations agree. Implicit declaration of functions is not allowed: every function must be explicitly declared before it can be called. In C89, if a function is called without an explicit prototype, the compiler provides an implicit declaration. The compiler checks for mismatches between the parameters of a function call and those in the function declaration. The compiler also uses the declaration for argument type checking and argument conversions.

A function *definition* contains a function declaration and the body of the function. A function can only have one definition.

Declarations are typically placed in header files, while function definitions appear in source files.



A *function argument* is an expression that you use within the parentheses of a function call. A *function parameter* is an object or reference declared within the parenthesis of a function declaration or definition. When you call a function, the arguments are evaluated, and each parameter is initialized with the value of the corresponding argument. The semantics of argument passing are identical to those of assignment.

Some declarations do not name the parameters within the parameter lists; the declarations simply specify the types of parameters and the return values. This is called *prototyping* A function prototype consists of the function return type, the name of the function, and the parameter list. The following example demonstrates this:

```
int func(int,long);
```

Function prototypes are required for compatibility between C and C++. The nonprototype form of a function that has an empty parameter list means that the function takes an unknown number of parameters in C, whereas in C++, it means that it takes no parameters.

Function Return Type

You can define a function to return any type of value, except an array type or a function type. These exclusions must be handled by returning a pointer to the array or function. A function may return a pointer to function, or a pointer to the first element of an array, but it may not return a value that has a type of array or function. To indicate that the function does not return a value, declare it with a return type of **void**.

The return type of a function must be **void** if the return statement does not contain an expression. However, if an expression does appear in the return statement, then the return type of the function cannot be **void**: the compiler converts the return expression as if by assignment to the return type of the function.

▶ C++ The return statement of a function does not require an expression if the function has the return type **void**, or is a constructor or destructor. In these cases, the function does not return a value. When a function returns a value, the return statement must contain an expression, which is returned to the caller of the function after being implicitly converted to the return type of the function in which it appears.

A function cannot be declared as returning a data object having a **volatile** or **const** type, but it can return a pointer to a **volatile** or **const** object.

▶ C++ Limitations When Declaring Functions in C++

Every function declaration must specify a return type.

Only member functions may have **const** or **volatile** specifiers after the parenthesized parameter list.

The *exception_specification* limits the function from throwing only a specified list of exceptions.

Other Limitations When Declaring a Function

The ellipsis (**...**) may be the only argument in C++. In this case, the comma is not required. In C, you cannot have an ellipsis as the only argument.

Types cannot be defined in return or argument types. For example, the C++ compiler allows the following declaration of `print()`:

```
struct X { int i; };
void print(X x);
```

The C compiler allows the following declaration:

```
struct X { int i; };
void print(struct X x);
```

Neither the C nor C++ compiler allows the following declaration of the same function:

```
void print(struct X { int i; } x);   //error
```

This example attempts to declare a function `print()` that takes an object `x` of class `X` as its argument. However, the class definition is not allowed within the argument list.

In another example, the C++ compiler will allow the following declaration of `counter()`:

```
enum count {one, two, three};
count counter();
```

Similarly, the C compiler allows the following declaration:

```
enum count {one, two, three};
enum count counter();
```

Neither compiler allows the following declaration of the same function:

```
enum count{one, two, three} counter();   //error
```

In the attempt to declare `counter()`, the enumeration type definition cannot appear in the return type of the function declaration.

Related References
- "Type Qualifiers" on page 71
- "Exception Specifications" on page 380

C++ Function Declarations

▶ C++ In C++, you can specify the qualifiers **volatile** and **const** in member function declarations. You can also specify exception specifications in function declarations. All C++ functions must be declared before they can be called.

Related References
- "Type Qualifiers" on page 71
- "const and volatile Member Functions" on page 266
- "Exception Specifications" on page 380

Multiple Function Declarations

▶ C++ All function declarations for a particular function must have the same number and type of parameters, and must have the same return type.

These return and parameter types are part of the function type, although the default arguments and exception specifications are not.

If a previous declaration of an object or function is visible in an enclosing scope, the identifier has the same linkage as the first declaration. However, a variable or function that has no linkage and later declared with a linkage specifier will have the linkage you have specified.

For the purposes of argument matching, ellipsis and linkage keywords are considered a part of the function type. They must be used consistently in all declarations of a function. If the only difference between the parameter types in two declarations is in the use of **typedef** names or unspecified argument array bounds, the declarations are the same. A **const** or **volatile** type specifier is also part of the function type, but can only be part of a declaration or definition of a nonstatic member function.

If two function declarations match in both return type and parameter lists, then the second declaration is treated as redeclaration of the first. The following example declares the same function:

```
int foo(const string &bar);
int foo(const string &);
```

Declaring two functions differing only in return type is not valid function overloading, and is flagged as a compile-time error. For example:

```
void f();
int f();      // error, two definitions differ only in
              // return type
int g()
{
   return f();
}
```

Related References
- "Overloading Functions" on page 237

Parameter Names in Function Declarations

C++ You can supply parameter names in a function declaration, but the compiler ignores them except in the following two situations:

1. If two parameter names have the same name within a single declaration. This is an error.
2. If a parameter name is the same as a name outside the function. In this case the name outside the function is hidden and cannot be used in the parameter declaration.

In the following example, the third parameter name `intersects` is meant to have enumeration type `subway_line`, but this name is hidden by the name of the first parameter. The declaration of the function `subway()` causes a compile-time error because `subway_line` is not a valid type name because the first parameter name `subway_line` hides the namespace scope **enum** type and cannot be used again in the second parameter.

```
enum subway_line {yonge,
university, spadina, bloor};
int subway(char * subway_line, int stations,
                  subway_line intersects);
```

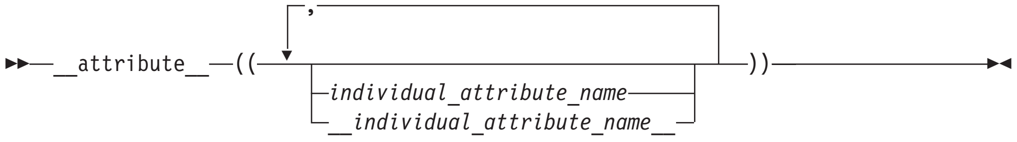
Function Attributes

Function attributes are orthogonal extensions, implemented to enhance the portability of programs developed with GNU C. Specifiable attributes for functions provide explicit ways to help the compiler optimize function calls and to instruct it to check more aspects of the code. Others provide additional functionality.

IBM C and C++ implement a subset of the GNU C function attributes. If a particular function attribute is not implemented, its specification is accepted and the semantics are ignored. These language features are collectively available when compiling in any of the extended language levels.

The IBM language extensions for function attributes preserve the GNU C syntax. A function attribute specification using the form *__attribute_name__* (that is, the function attribute keyword with double underscore characters leading and trailing) reduces the likelihood of a name conflict with a macro of the same name.

The keyword **__attribute__** introduces an attribute specifier. Some of the attributes can also be applied to variables. The syntax is of the general form:



Function attributes are attached to a declarator.

C For attributes specified on a function prototype declaration, attaching them to the declarator means placing them after the closing parenthesis of the parameter list.

```
/* Specify the attribute on a function prototype declaration */
void f(int i, int j) __attribute__((individual_attribute_name));
void f(int i, int j) { }
```

C Due to ambiguities in parsing old-style parameter declarations, a function definition must have the attribute specification precede the declarator. For example, the following definitions of foo show the correct placement:

```
int __attribute__((individual_attribute_name)) foo(int i) { }
int __attribute__((individual_attribute_name)) foo(i,j)
  int i; int j;
{ }
```

C++ For C++, function attributes are placed after the declarator on either declarations or definitions. For typical functions, this is also after the closing parenthesis; however, function attributes must follow any exception specification that may be present for the function.

Related References

- “Variable Attributes” on page 32
- “Type Attributes” on page 45

The alias Function Attribute

AIX Linux z/OS The **alias** function attribute causes the function declaration to appear in the object file as an alias for another symbol. This language feature provides a technique for coping with duplicate or cumbersome names.

The **alias** function attribute follows the general syntax for function attributes. The following diagram shows the supported forms.

```
►►──__attribute__──((──┬─alias─────┬──(──"original_function_name"──)──))──────────►◄
                       └─__alias__─┘
```

C The aliased function can be defined after the specification of its alias with this function attribute. C also allows an alias specification in the absence of a definition of the aliased function in the same compilation unit.

C++ The *original_function_name* must be the mangled name.

The following declares bar to be an alias for __foo:

C

```
void __foo(){   /*  function body  */   }
void bar() __attribute__((alias("__foo")));
```

C++

```
extern "C" __foo(){   /*  function body  */   }
void bar() __attribute__((alias("__foo")));
```

The compiler does not check for consistency between the declaration of bar and definition of __foo. Such consistency remains the responsibility of the programmer.

Related References
- "The weak Function Attribute" on page 169

The always_inline Function Attribute

Function attribute always_inline instructs the compiler to inline an **inline** function, regardless of whether optimization was specified at compile time. However, the attribute has no effect if the program is compiled at no-opt levels. Specifying this attribute for a function without an **inline** specification also has no effect. The attribute takes precedence over inlining compiler options. The language feature is an orthogonal extension to C89, C99, Standard C++ and C++98, and has been implemented to facilitate porting programs developed with GNU C and C++.

The syntax is shown in the following diagram.

```
►►──__attribute__──((──┬─always_inline─────┬──))──────────────────────────────────►◄
                       └─__always_inline__─┘
```

The const Function Attribute

The **const** function attribute allows you to tell the compiler that the function can safely be called fewer times than indicated in the source code. The language feature provides the programmer with an explicit way to help the compiler optimize code by indicating that the function does not examine any values except its arguments and has no effects except for its return value.

The **const** function attribute follows the general syntax for function attributes.

```
►►──__attribute__──((──┬─const─────┬─))──────────────────────────────────►◄
                       └─__const__─┘
```

The following kinds of functions should not be declared **const**:
- A function with pointer arguments which examines the data pointed to.
- A function that calls a non-**const** function.

See also `#pragma isolated_call` in *XL C/C++ Compiler Reference*.

The constructor and destructor Function Attributes

Linux The **constructor** and **destructor** function attributes provide the ability to write a function that initializes data or releases storage that is used implicitly during program execution. A function to which the **constructor** function attribute has been applied is called automatically before execution enters `main()`. Similarly, a function to which the **destructor** attribute has been applied is called automatically after calling `exit()` or upon completion of `main()`.

When the constructor or destructor function is called automatically, the return value of the function is ignored, and any parameters of the function are undefined.

The constructor and destructor function attributes follow the general syntax for function attributes: in the prototype declaration, the attribute specifier follows the function declarator; in the function definition, it precedes the declarator. C++ does not accept the attribute before the function declarator in function definitions. The following diagram shows the supported forms of the function attribute.

```
►►──__attribute__──((──┬─constructor─────┬─))─function_name─(─arguments─)──────►◄
                       ├─destructor──────┤
                       ├─__constructor__─┤
                       └─__destructor__──┘
```

A function declaration containing a constructor or destructor function attribute must match all of its other declarations.

The format Function Attribute

Function attribute `format` provides a way to identify user-defined functions that take format strings as arguments so that calls to these functions will be type-checked against a format string, similar to the way the compiler checks calls to the functions `printf`, `scanf`, `strftime`, and `strfmon` for errors. The feature is an orthogonal extension to C89, C99, Standard C++ and C++98, and has been implemented to facilitate porting applications developed with GNU C and C++.

The syntax is shown in the following diagram. The first argument indicates the archetype for how the format string should be interpreted.

```
                     ┌─,──────────────────────────────────────────────────────────────────┐
                     ▼                                                                    │
►►──__attribute__──((───┬─format─────┬─(─┬─printf─────┬─,─string_index─,─first_to_check─)─┴─))──────►◄
                        └─__format__─┘   ├─scanf──────┤
                                         ├─strftime───┤
                                         ├─strfmon────┤
                                         ├─__printf__─┤
                                         ├─__scanf__──┤
                                         ├─__strftime__┤
                                         └─__strfmon__─┘
```

where

string_index
: Is a constant integral expression that specifies which argument in the declaration of the user function is the format string argument. In C++, the minimum value of *string_index* for nonstatic member functions is 2 because the first argument is an implicit **this** argument. This behavior is consistent with that of GNU C++.

first_to_check
: Is a constant integral expression that specifies the first argument to check against the format string. If there are no arguments to check against the format string (that is, diagnostics should only be performed on the format string syntax and semantics), *first_to_check* should have a value of `0`. For `strftime`-style formats, *first_to_check* is required to be `0`.

It is possible to specify multiple `format` attributes on the same function, in which case, all apply.

```
void my_fn(const char* a, const char* b, ...)
      __attribute__((__format__(__printf__,1,0), __format__(__scanf__,2,3)));
```

It is also possible to diagnose the same string for different format styles. All styles are diagnosed.

```
void my_fn(const char* a, const char* b, ...)
      __attribute__((__format__(__printf__,2,3),
                     __format__(__strftime__,2,0),
                     __format__(__scanf__,2,3)));
```

The format_arg Function Attribute

Function attribute `format_arg` provides a way to identify user-defined functions that modify format strings. Once the function is identified, calls to functions like `printf`, `scanf`, `strftime`, or `strfmon`, whose operands are a call to the user-defined function can be checked for errors. The language feature is an orthogonal extension to C89, C99, and Standard C++, and has been implemented to facilitate porting programs developed with GNU C and C++.

The syntax is shown in the following diagram.

```
►►──__attribute__──((──┬─format_arg─────┬──(──string_index──)──))──────────────►◄
                       └─__format_arg__─┘
```

where *string_index* is a constant integral expression that specifies which argument is the format string argument, starting from 1. For non-static member functions in C++, *string_index* starts from `2` because the first parameter is an implicit **this** parameter.

It is possible to specify multiple `format_arg` attributes on the same function, in which case, all apply.

```
extern char* my_dgettext(const char* my_format, const char* my_format2)
         __attribute__((__format_arg__(1))) __attribute__((__format_arg__(2)));

printf(my_dgettext("%","%"));
//printf-style format diagnostics are performed on both "%" strings
```

The noinline Function Attribute

Function attribute `noinline` prevents the function to which it is applied from being inlined, regardless if the function is declared inline or non-inline. The attribute takes precedence over inlining compiler options, the **inline** keyword, and the `always_inline` function attribute. The language feature is an orthogonal extension to C89, C99, Standard C++ and C++98, and has been implemented to facilitate porting programs developed with GNU C and C++.

The syntax is shown in the following diagram.

```
►►──__attribute__──((──┬─noinline─────┬─))──────────────────────────────────►◄
                       └─__noinline__─┘
```

Other than preventing inlining, the attribute does not remove the semantics of inline functions.

The noreturn Function Attribute

The **noreturn** function attribute allows you to indicate to the compiler that the function is not intended to return. The language feature provides the programmer with another explicit way to help the compiler optimize code and to reduce false warnings for uninitialized variables.

The return type of the function should be **void**.

The **noreturn** function attribute follows the general syntax for function attributes.

```
►►──__attribute__──((──┬─noreturn─────┬─))──────────────────────────────────►◄
                       └─__noreturn__─┘
```

Registers saved by the calling function may not necessarily be restored before calling the nonreturning function.

See also `#pragma leaves` in *XL C/C++ Compiler Reference*.

The pure Function Attribute

The function attribute **pure** allows you to declare a function that can be called fewer times than what is literally in the source code. Declaring a function with the attribute **pure** indicates that the function has no effect except a return value that depends only on the parameters, global variables, or both. The syntax is the same as that for **const**.

See also `#pragma isolated_call` in *XL C/C++ Compiler Reference*.

The section Function Attribute

▷ Linux The **section** function attribute specifies the section in the object file in which the compiler should place its generated code. The language feature provides the ability to control the section in which a function should appear.

The section function attribute follows the general syntax for function attributes: in the prototype declaration, the attribute specifier follows the function declarator; in the function definition, it precedes the declarator. C++ does not accept the attribute before the function declarator in function definitions. The following diagram shows the supported forms of the function attribute.

```
►►──__attribute__──((──┬─section─────┬─(──"section_name"──)──))───────────►◄
                       └─__section__─┘
```

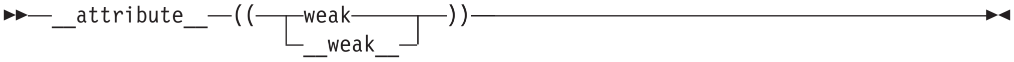
where *section_name* is a string literal.

Each defined function can reside in only one section. The section indicated in a function definition should match that in any previous declaration. The section indicated in a function definition cannot be overwritten, whereas one in a function declaration can be overwritten by a later specification. Moreover, if a section attribute is applied to a function declaration, the function will be placed in the specified section only if it is defined in the same compilation unit.

The weak Function Attribute

AIX Linux The **weak** function attribute causes the symbol resulting from the function declaration to appear in the object file as a weak symbol, rather than a global one. The **weak** attribute can also be applied to variables. The language feature provides the programmer writing library functions with a way to preempt duplicate name errors if the user overrides the function definition in his or her code.

The **weak** function attribute follows the general syntax for function attributes. The following diagram shows the supported forms.



Normally, when several relocatable object files are processed, the linker disallows multiple definitions of global symbols with the same name. However, the linker allows a weak definition in the presence of a global symbol with the same name; the weak definition is ignored. Another difference between a global and a weak symbol lies in whether the linker searches archive libraries. To resolve undefined global symbols, the linker searches archive libraries and extracts members that contain definitions; it does not do this to resolve undefined weak symbols.

The following restrictions and limitations apply to weak symbols:

- Weak symbols may not have `static` storage duration.
- Multiple definitions for a weak symbol cannot be provided in the same translation unit. When multiple definitions are present, the linker uses the first weak definition encountered.

Examples of Function Declarations

The following code fragments show several function declarations. The first declares a function `f` that takes two integer arguments and has a return type of **void**:

```
void f(int, int);
```

The following code fragment declares a pointer `p1` to a function that takes a pointer to a constant character and returns an integer:

```
int (*p1) (const char*);
```

The following code fragment declares a function `f1` that takes an integer argument, and returns a pointer to a function that takes an integer argument and returns an integer:

```
int (*f1(int)) (int);
```

Alternatively, a **typedef** can be used for the complicated return type of function `f1`:

```
typedef int f1_return_type(int);
f1_return_type* f1(int);
```

C++ The remainder of this section pertains to C++ only.

The following declaration is of an external function `f2` that takes a constant integer as its first argument, can have a variable number and variable types of other arguments, and returns type **int**.

```
int extern f2(const int ...);
```

However, in C, a comma is required before the ellipsis:

```
int extern f2(const int, ...);
```

Function `f3` has a return type **int**, and takes an **int** argument with a default value that is the value returned from function `f2`:

```
const int j = 5;
int f3( int x = f2(j) );
```

Function `f6` is a **const** class member function of class `X`, takes no arguments, and has a return type of **int**:

```
class X
{
public:
      int f6() const;
};
```

Function `f4` takes no arguments, has return type **void**, and can throw class objects of types `X` and `Y`.

```
class X;
class Y;

// ...

void f4() throw(X,Y);
```

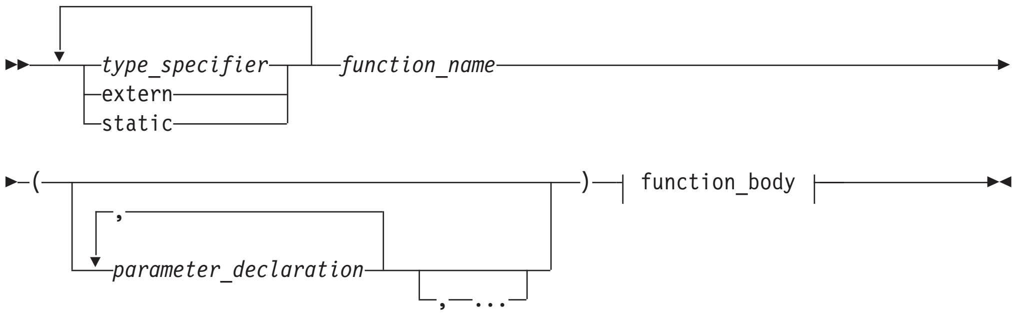
Function `f5` takes no arguments, has return type **void**, and will call `unexpected()` if it throws an exception of any type.

```
void f5() throw();
```

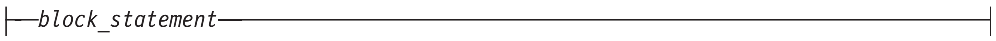
Function Definitions

A *function definition* contains a function declaration and the body of a function.

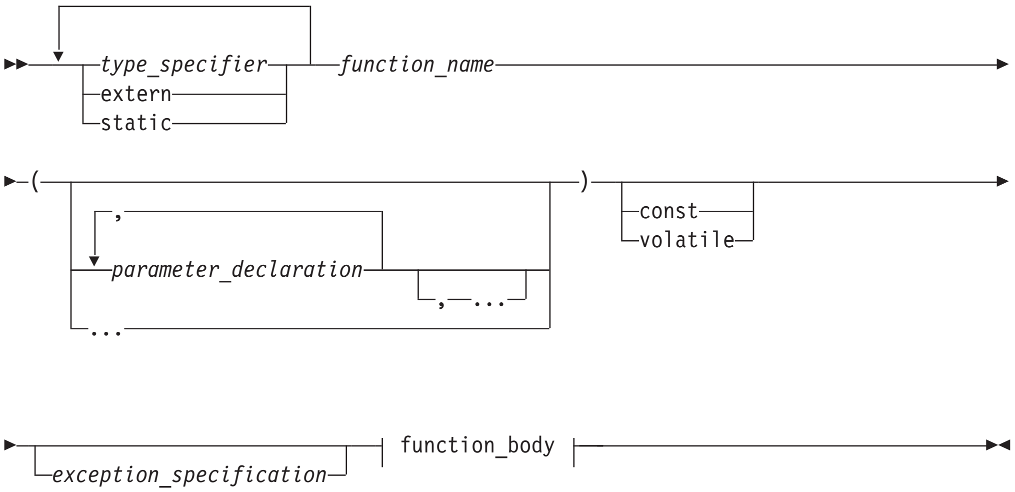
C The syntax for a C function definition is as follows:



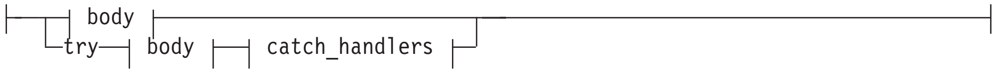
function_body:



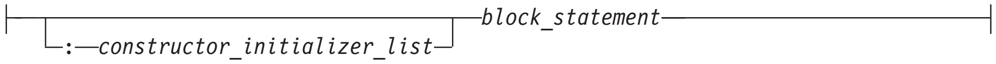
C++ The syntax for a C++ function definition is as follows:



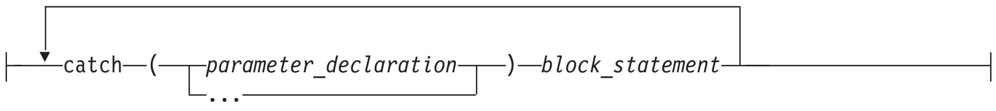
function_body:



body:



catch_handlers:



In both languages, a function definition contains the following:

- At least one *type specifier*, which determines the type of value that the function returns. For example, the syntax for a function that returns an **unsigned long int** uses three type specifiers.
- An optional *storage class specifier* **extern** or **static**, which determines the scope of the function. If a storage class specifier is not given, the function has external linkage.

- A *function declarator* is the function name followed by a parenthesized list of parameter types and names each parameter that the function expects. In the following function definition, `f(int a, int b)` is the function declarator:

  ```
  int f(int a, int b) {
    return a + b;
  }
  ```
- A *block statement*, which contains data definitions and code.

▶ C++ A C++ function definition may optionally contain the following:

- **const** or **volatile** specifiers after the function declarator for a member function.
- An *exception specification*, which limits the function from throwing only a specified list of exceptions.
- A *try block* with one or more *catch handlers* instead of a *block statement*.
- A *constructor initializer list* before the block statement if the function definition is for a constructor. In the following definition of class `A`, `x(0), y('c')` is the constructor initializer list:

  ```
  class A {
    int x;
    char y;
  public:
    A() : x(0), y('c') { }
  };
  ```

A function can be called by itself or by other functions. By default, function definitions have external linkage, and can be called by functions defined in other files. A storage class specifier of **static** means that the function name has global scope only, and can be directly invoked only from within the same translation unit.

▶ C++ This use of **static** is deprecated in C++. Instead, place the function in the unnamed namespace.

▶ C In C, if a function definition has external linkage and a return type of **int**, calls to the function can be made before it is visible because an implicit declaration of `extern int func();` is assumed. To be compatible with C++, all functions must be declared with prototypes.

▶ C If the function does not return a value, use the keyword **void** as the type specifier. If the function does not take any parameters, use the keyword **void** rather than an empty parameter list to indicate that the function is not passed any arguments. In C, a function with an empty parameter list signifies a function that takes an unknown number of parameters; in C++, it means it takes no parameters.

▶ C In C, you cannot declare a function as a struct or union member.

Compatibility of Function Declarations

All declarations for a given function must be compatible; that is, the return type is the same and the parameters have the same type.

Compatibility of Function Types

▶ C The notion of type compatibility pertains only to C. For two function types to be compatible, the return types must be compatible. If both function types are specified without prototypes, this is the only requirement.

For two functions declared with prototypes, the composite type must meet the following additional requirements:

- If one of the function types has a parameter type list, the composite type is a function prototype with the same parameter type list.
- If both types are function types with parameter lists, then each parameter in the parameter list of the composite is the composite type of the corresponding parameters.

and may use the [*] notation in their sequences of declarator specifiers to specify variable length array types.

If the function declarator is not part of the function definition, the parameters may have incomplete type. The parameters may also specify variable length array types by using the [*] notation in their sequences of declarator specifiers. The following are examples of compatible function prototype declarators:

```
double maximum(int n, int m, double a[n][m]);
double maximum(int n, int m, double a[*][*]);
double maximum(int n, int m, double a[ ][*]);
double maximum(int n, int m, double a[ ][m]);
```

Examples of Function Definitions

The following example is a definition of the function sum:

```
int sum(int x,int y)
{
   return(x + y);
}
```

The function sum has external linkage, returns an object that has type **int**, and has two parameters of type **int** declared as x and y. The function body contains a single statement that returns the sum of x and y.

In the following example, ary is an array of two function pointers. Type casting is performed to the values assigned to ary for compatibility:

```
#include <stdio.h>

typedef void (*ARYTYPE)();

int func1(void);
void func2(double a);

int main(void)
{
  double num = 333.3333;
  int retnum;
  ARYTYPE ary[2];
  ary[0]=(ARYTYPE)func1;
  ary[1]=(ARYTYPE)func2;

  retnum=((int (*)())ary[0])();      /*  calls func1  */
  printf("number returned = %i\n", retnum);
  ((void (*)(double))ary[1])(num);   /*  calls func2  */

  return(0);
}

int func1(void)
{
  int number=3;
  return number;
}
```

```
void func2(double a)
{
  printf("result of func2 = %f\n", a);
}
```

The following is the output of the above example:

```
number
returned = 3
result of func2 = 333.333300
```

Related References

- “extern Storage Class Specifier” on page 38
- “static Storage Class Specifier” on page 40
- “Type Qualifiers” on page 71

Ellipsis and void

An ellipsis at the end of the parameter specifications is used to specify that a function has a variable number of parameters. The number of parameters is equal to, or greater than, the number of parameter specifications. At least one parameter declaration must come before the ellipsis.

```
int f(int, ...);
```

C++ The comma before the ellipsis is optional. In addition, a parameter declaration is not required before the ellipsis.

C The comma before the ellipsis as well as a parameter declaration before the ellipsis are both required in C.

Parameter promotions are performed as needed, but no type checking is done on the variable arguments.

You can declare a function with no arguments in two ways:

```
int f(void);
int f();
```

C++ An empty argument declaration list or the argument declaration list of `(void)` indicates a function that takes no arguments.

C An empty argument declaration list means that the function may take any number or type of parameters.

The type **void** cannot be used as an argument type, although types derived from **void** (such as pointers to **void**) can be used.

In the following example, the function `f()` takes one integer argument and returns no value, while `g()` expects no arguments and returns an integer.

```
void f(int);
int g(void);
```

Examples of Function Definitions

The following example contains a function declarator `i_sort` with `table` declared as a pointer to **int** and `length` declared as type **int**. Note that arrays as parameters are implicitly converted to a pointer to the element type.

```
/**
 ** This example illustrates function definitions.
 ** Note that arrays as parameters are implicitly
 ** converted to a pointer to the type.
 **/

#include <stdio.h>

void i_sort(int table[ ], int length);

int main(void)
{
   int table[ ]={1,5,8,4};
   int length=4;
   printf("length is %d\n",length);
   i_sort(table,length);
}
void i_sort(int table[ ], int length)
{
  int i, j, temp;

  for (i = 0; i < length -1; i++)
    for (j = i + 1; j < length; j++)
      if (table[i] > table[j])
      {
        temp = table[i];
        table[i] = table[j];
        table[j] = temp;
      }
}
```

The following are examples of function declarations (also called *function prototypes*):

```
double square(float x);
int area(int x,int y);
static char *search(char);
```

The following example illustrates how a **typedef** identifier can be used in a function declarator:

```
typedef struct tm_fmt { int minutes;
                      int hours;
                      char am_pm;
                    } struct_t;
long time_seconds(struct_t arrival)
```

The following function set_date declares a pointer to a structure of type date as a parameter. date_ptr has the storage class specifier **register**.

```
void set_date(register struct date *date_ptr)
{
  date_ptr->mon = 12;
  date_ptr->day = 25;
  date_ptr->year = 87;
}
```

▶ C C99 requires at least one type specifier for each parameter in a declaration, which reduces the number of situations where the compiler behaves as if an implicit **int** were declared. Prior to C99, the type of b or c in the declaration of foo is ambiguous, and the compiler would assume an implicit **int** for both.

```
int foo( char a, b, c )
{
   /* statements */
}
```

The main() Function

When a program begins running, the system calls the function `main`, which marks the entry point of the program. Every program must have one function named `main`. No other function in the program can be called `main`. A `main` function has one of two forms:

C `int main (void)` *block_statement*

C++ `int main ( )`*block_statement*

`int main (int argc, char ** argv)`*block_statement*

The argument `argc` is the number of command-line arguments passed to the program. The argument `argv` is a pointer to an array of strings, where `argv[0]` is the name you used to run your program from the command-line, `argv[1]` the first argument that you passed to your program, `argv[2]` the second argument, and so on.

By default, `main` has the storage class **extern**.

C++ You cannot declare `main` as **inline** or **static**. You cannot call `main` from within a program or take the address of `main`. You cannot overload this function.

Arguments to main

The function `main` can be declared with or without parameters.

```
int main(int argc, char *argv[])
```

Although any name can be given to these parameters, they are usually referred to as `argc` and `argv`.

The first parameter, `argc` (argument count), has type **int** and indicates how many arguments were entered on the command line.

The second parameter, `argv` (argument vector), has type array of pointers to **char** array objects. **char** array objects are null-terminated strings.

The value of `argc` indicates the number of pointers in the array `argv`. If a program name is available, the first element in `argv` points to a character array that contains the program name or the invocation name of the program that is being run. If the name cannot be determined, the first element in `argv` points to a null character.

This name is counted as one of the arguments to the function `main`. For example, if only the program name is entered on the command line, `argc` has a value of 1 and `argv[0]` points to the program name.

Regardless of the number of arguments entered on the command line, `argv[argc]` always contains `NULL`.

Example of Arguments to main

The following program `backward` prints the arguments entered on a command line such that the last argument is printed first:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  while (--argc > 0)
    printf("%s ", argv[argc]);
}
```

Invoking this program from a command line with the following:

```
backward string1 string2
```

gives the following output:

```
string2 string1
```

The arguments argc and argv would contain the following values:

Object	Value
argc	3
argv[0]	pointer to string "backward"
argv[1]	pointer to string "string1"
argv[2]	pointer to string "string2"
argv[3]	NULL

Note: Be careful when entering mixed case characters on a command line because some environments are not case-sensitive. Also, the exact format of the string pointed to by argv[0] is system-dependent.

Calling Functions and Passing Arguments

The arguments of a function call are used to initialize the parameters of the function definition. Array expressions and C function designators as arguments are converted to pointers before the call.

Integral and floating-point promotions will first be done to the values of the arguments before the function is called.

The type of an argument is checked against the type of the corresponding parameter in the function declaration. The size expressions of each variably modified parameter are evaluated on entry to the function. All standard and user-defined type conversions are applied as necessary. The value of each argument expression is converted to the type of the corresponding parameter as if by assignment.

For example:

```
#include <stdio.h>
#include <math.h>

/* Declaration */
extern double root(double, double);

/* Definition */
double root(double value, double base) {
  double temp = exp(log(value)/base);
  return temp;
}

int main(void) {
  int value = 144;
  int base = 2;
  printf("The root is: %f\n", root(value, base));
  return 0;
}
```

The output is `The root is: 12.000000`

In the above example, because the function `root` is expecting arguments of type **double**, the two **int** arguments `value` and `base` are implicitly converted to type **double** when the function is called.

The order in which arguments are evaluated and passed to the function is implementation-defined. For example, the following sequence of statements calls the function `tester`:

```
int x;
x = 1;
tester(x++, x);
```

The call to `tester` in the example may produce different results on different compilers. Depending on the implementation, `x++` may be evaluated first or `x` may be evaluated first. To avoid the ambiguity and have `x++` evaluated first, replace the preceding sequence of statements with the following:

```
int x, y;
x = 1;
y = x++;
tester(y, x);
```

▶ C++ The remainder of this section pertains only to C ++.

If a nonstatic class member function is passed as an argument, the argument is converted to a pointer to member.

If a function parameter is a variable length array, the array dimensions other than the leftmost dimension distinguish the overload set for that function.

If a class has a destructor or a copy constructor that does more than a bitwise copy, passing a class object by value results in the construction of a temporary that is actually passed by reference.

It is an error when a function argument is a class object and all of the following properties hold:
- The class needs a copy constructor.
- The class does not have a user-defined copy constructor.
- A copy constructor cannot be generated for that class.

Passing Arguments by Value

If you call a function with an argument that corresponds to a non-reference parameter, you have passed that argument by value. The parameter is initialized with the value of the argument. You can change the value of the parameter (if that parameter has not been declared **const**) within the scope of the function, but these changes will not affect the value of the argument in the calling function.

The following are examples of passing arguments by value:

The following statement calls the function **printf**, which receives a character string and the return value of the function `sum`, which receives the values of `a` and `b`:

```
printf("sum = %d\n", sum(a,b));
```

The following program passes the value of `count` to the function `increment`, which increases the value of the parameter `x` by 1.

```
/**
 ** An example of passing an argument to a function
 **/

#include <stdio.h>

void increment(int);

int main(void)
{
  int count = 5;

  /* value of count is passed to the function */
  increment(count);
  printf("count = %d\n", count);

  return(0);
}

void increment(int x)
{
  ++x;
  printf("x = %d\n", x);
}
```

The output illustrates that the value of `count` in `main` remains unchanged:

```
x = 6
count = 5
```

Related References
- "Function Call Operator ( )" on page 109

Passing Arguments by Reference

Passing by reference refers to a method of passing arguments where the value of an argument in the calling function can be modified in the called function.

C To pass an argument by reference, you declare the corresponding parameter with a pointer type.

C++ To pass an argument by reference in C++, the corresponding parameter can be any reference type, not just pointer types.

The following example shows how arguments are passed by reference. Note that reference parameters are initialized with the actual arguments when the function is called.

```
#include <stdio.h>

void swapnum(int &i, int &j) {
  int temp = i;
  i = j;
  j = temp;
}

int main(void) {
  int a = 10;
  int b = 20;
  swapnum(a, b);
  printf("A is %d and B is %d\n", a, b);
  return 0;
}
```

When the function swapnum() is called, the actual values of the variables a and b are exchanged because they are passed by reference. The output is:

```
A is 20 and B is 10
```

You must define the parameters of swapnum() as references if you want the values of the actual arguments to be modified by the function swapnum().

C++ In order to modify a reference that is **const**-qualified, you must cast away its constness with the **const_cast** operator. The following example demonstrates this:

```
#include <iostream>
using namespace std;

void f(const int& x) {
  int* y = const_cast<int>(&x);
  (*y)++;
}

int main() {
  int a = 5;
  f(a);
  cout << a << endl;
}
```

This example outputs 6.

You can modify the values of nonconstant objects through pointer parameters. The following example demonstrates this:

```
#include <stdio.h>

int main(void)
{
  void increment(int *x);
  int count = 5;

  /* address of count is passed to the function */
  increment(&count);
  printf("count = %d\n", count);

  return(0);
}

void increment(int *x)
{
  ++*x;
  printf("*x = %d\n", *x);
}
```

The following is the output of the above code:

```
*x = 6
count = 6
```

The example passes the address of count to increment(). Function increment() increments count through the pointer parameter x.

Default Arguments in C++ Functions

C++ You can provide default values for function parameters. For example:

```
#include <iostream>
using namespace std;

int a = 1;
int f(int a) { return a; }
int g(int x = f(a)) { return x; }

int h() {
  a = 2;
  {
    int a = 3;
    return g();
  }
}

int main() {
  cout << h() << endl;
}
```

This example prints `2` to standard output, because the `a` referred to in the declaration of `g()` is the one at file scope, which has the value `2` when `g()` is called.

The default argument must be implicitly convertible to the parameter type.

A pointer to a function must have the same type as the function. Attempts to take the address of a function by reference without specifying the type of the function will produce an error. The type of a function is not affected by arguments with default values.

The following example shows that default arguments are not considered part of a function's type. The default argument allows you to call a function without specifying all of the arguments, it does not allow you to create a pointer to the function that does not specify the types of all the arguments. Function `f` can be called without an explicit argument, but the pointer `badpointer` cannot be defined without specifying the type of the argument:

```
int f(int = 0);
void g()
{
   int a = f(1);                // ok
   int b = f();                 // ok, default argument used
}
int (*pointer)(int) = &f;       // ok, type of f() specified (int)
int (*badpointer)() = &f;       // error, badpointer and f have
                                // different types. badpointer must
                                // be initialized with a pointer to
                                // a function taking no arguments.
```

Related References

- "Pointers to Functions" on page 185

Restrictions on Default Arguments

Of the operators, only the function call operator and the operator **new** can have default arguments when they are overloaded.

Parameters with default arguments must be the trailing parameters in the function declaration parameter list. For example:

```
void f(int a, int b = 2, int c = 3);  // trailing defaults
void g(int a = 1, int b = 2, int c);  // error, leading defaults
void h(int a, int b = 3, int c);      // error, default in middle
```

Once a default argument has been given in a declaration or definition, you cannot redefine that argument, even to the same value. However, you can add default arguments not given in previous declarations. For example, the last declaration below attempts to redefine the default values for a and b:

```
void f(int a, int b, int c=1);     // valid
void f(int a, int b=1, int c);     // valid, add another default
void f(int a=1, int b, int c);     // valid, add another default
void f(int a=1, int b=1, int c=1); // error, redefined defaults
```

You can supply any default argument values in the function declaration or in the definition. Any parameters in the parameter list following a default argument value must have a default argument value specified in this or a previous declaration of the function.

You cannot use local variables in default argument expressions. For example, the compiler generates errors for both function g() and function h() below:

```
void f(int a)
{
      int b=4;
      void g(int c=a); // Local variable "a" cannot be used here
      void h(int d=b); // Local variable "b" cannot be used here
}
```

Related References
- "Function Call Operator ( )" on page 109
- "C++ new Operator" on page 126
- "Default Arguments in C++ Functions" on page 181

Evaluating Default Arguments

When a function defined with default arguments is called with trailing arguments missing, the default expressions are evaluated. For example:

```
void f(int a, int b = 2, int c = 3); // declaration
// ...
int a = 1;
f(a);            // same as call f(a,2,3)
f(a,10);         // same as call f(a,10,3)
f(a,10,20);      // no default arguments
```

Default arguments are checked against the function declaration and evaluated when the function is called. The order of evaluation of default arguments is undefined. Default argument expressions cannot use other parameters of the function. For example:

```
int f(int q = 3, int r = q); // error
```

The argument r cannot be initialized with the value of the argument q because the value of q may not be known when it is assigned to r. If the above function declaration is rewritten:

```
int q=5;
int f(int q = 3, int r = q); // error
```

The value of r in the function declaration still produces an error because the variable q defined outside of the function is hidden by the argument q declared for the function. Similarly:

```
typedef double D;
int f(int D, int z = D(5.3) ); // error
```

Here the type D is interpreted within the function declaration as the name of an integer. The type D is hidden by the argument D. The cast D(5.3) is therefore not interpreted as a cast because D is the name of the argument not a type.

In the following example, the nonstatic member a cannot be used as an initializer because a does not exist until an object of class X is constructed. You can use the static member b as an initializer because b is created independently of any objects of class X. You can declare the member b after its use as a default argument because the default values are not analyzed until after the final bracket } of the class declaration.

```
class X
{
   int a;
    f(int z = a) ; // error
    g(int z = b) ; // valid
    static int b;
};
```

Related References
- "Default Arguments in C++ Functions" on page 181

Function Return Values

You must return a value from a function unless the function has a return type of **void**.

The return value is specified in a **return** statement. The following code fragment shows a function definition, including the **return** statement:

```
int add(int i, int j)
{
  return i + j; // return statement
}
```

The function add() can be called as shown in the following code fragment:

```
int a = 10,
    b = 20;
int answer = add(a, b); // answer is 30
```

In this example, the return statement initializes a variable of the returned type. The variable answer is initialized with the **int** value 30. The type of the returned expression is checked against the returned type. All standard and user-defined conversions are performed as necessary.

Each time a function is called, new copies of its variables with automatic storage are created. Because the storage for these automatic variables may be reused after the function has terminated, a pointer or reference to an automatic variable should not be returned.

▶ C++ If a class object is returned, a temporary object may be created if the class has copy constructors or a destructor.

Related References
- "return Statement" on page 204
- "Value of a return Expression and Function Value" on page 205

- "Temporary Objects" on page 327

Using References as Return Types

References can also be used as return types for functions. The reference returns the lvalue of the object to which it refers. This allows you to place function calls on the left side of assignment statements.

▶ C++ Referenced return values are used when assignment operators and subscripting operators are overloaded so that the results of the overloaded operators can be used as actual values.

Note: Returning a reference to an automatic variable gives unpredictable results.

Allocation and Deallocation Functions

▶ C++ You may define your own new operator or allocation function as a class member function or a global namespace function with the following restrictions:

- The first parameter must be of type **std::size_t**. It cannot have a default parameter.
- The return type must be of type **void***.
- Your allocation function may be a template function. Neither the first parameter nor the return type may depend on a template parameter.
- If you declare your allocation function with the empty exception specification `throw()`, your allocation function must return a null pointer if your function fails. Otherwise, your function must throw an exception of type **std::bad_alloc** or a class derived from **std::bad_alloc** if your function fails.

You may define your own delete operator or deallocation function as a class member function or a global namespace function with the following restrictions:

- The first parameter must be of type **void***.
- The return type must be of type **void**.
- Your deallocation function may be a template function. Neither the first parameter nor the return type may depend on a template parameter.

The following example defines replacement functions for global namespace **new** and **delete**:

```
#include <cstdio>
#include <cstdlib>

using namespace std;

void* operator new(size_t sz) {
  printf("operator new with %d bytes\n", sz);
  void* p = malloc(sz);
  if (p == 0) printf("Memory error\n");
  return p;
}

void operator delete(void* p) {
  if (p == 0) printf ("Deleting a null pointer\n");
  else {
    printf("delete object\n");
    free(p);
  }
}

struct A {
  const char* data;
  A() : data("Text String") { printf("Constructor of S\n"); }
  ~A() { printf("Destructor of S\n"); }
};

int main() {
  A* ap1 = new A;
  delete ap1;

  printf("Array of size 2:\n");
  A* ap2 = new A[2];
  delete[] ap2;
}
```

The following is the output of the above example:

```
operator
new with 40 bytes
operator new with 33 bytes
operator new with 4 bytes
Constructor of S
Destructor of S
delete object
Array of size 2:
operator new with 16 bytes
Constructor of S
Constructor of S
Destructor of S
Destructor of S
delete object
```

Related References

- “Free Store” on page 323

Pointers to Functions

A pointer to a function points to the address of the executable code of the function. You can use pointers to call functions and to pass functions as arguments to other functions. You cannot perform pointer arithmetic on pointers to functions.

The type of a pointer to a function is based on both the return type and parameter types of the function.

A declaration of a pointer to a function must have the pointer name in parentheses. The function call operator () has a higher precedence than the dereference operator *. Without them, the compiler interprets the statement as a function that returns a pointer to a specified return type. For example:

```
int *f(int a);       /* function f returning an int*                      */
int (*g)(int a);     /* pointer g to a function returning an int          */
char (*h)(int, int)  /* h is a function
                            that takes two integer parameters and returns char */
```

In the first declaration, f is interpreted as a function that takes an **int** as argument, and returns a pointer to an **int**. In the second declaration, g is interpreted as a pointer to a function that takes an **int** argument and that returns an **int**.

Related References

- “Pointer Conversions” on page 152

Inline Functions

An inline function is one for which the compiler copies the code from the function definition directly into the code of the calling function rather than creating a separate set of instructions in memory. Instead of transferring control to and from the function code segment, a modified copy of the function body may be substituted directly for the function call. In this way, the performance overhead of a function call is avoided.

A function is declared inline by using the **inline** function specifier or by defining a member function within a class or structure definition. The **inline** specifier is only a suggestion to the compiler that an inline expansion can be performed; the compiler is free to ignore the suggestion.

The following code fragment shows an inline function definition.

```
inline int add(int i, int j) { return i + j; }
```

The use of the **inline** specifier does not change the meaning of the function. However, the inline expansion of a function may not preserve the order of evaluation of the actual arguments. Inline expansion also does not change the linkage of a function: the linkage is external by default.

C++ In C++, both member and nonmember functions can be inlined. Member functions that are implemented inside the body of a class declaration are implicitly declared inline. Constructors, copy constructors, assignment operators, and destructors that are created by the compiler are also implicitly declared inline. An inline function that the compiler does not inline is treated similarly to an ordinary function: only a single copy of the function exists, regardless of the number of translation units in which it is defined.

C In C, any function with internal linkage can be inlined, but a function with external linkage is subject to restriction. The restrictions are as follows:

- If the **inline** keyword is used in the function declaration, then the function definition must appear in the same translation unit.
- An *inline definition* of a function is one in which all of the file-scope declarations for it in the same translation unit include the **inline** specifier without **extern**.
- An inline definition does not provide an external definition for the function: an external definition may appear in another translation unit. The inline definition serves as an alternative to the external definition when called from within the same translation unit. The C99 Standard does not prescribe whether the inline or external definition is used.

In C, an inline definition is distinct from the corresponding external definition and from any other corresponding inline definitions in other translation units.

When source code is compiled allowing language extensions, the behavior of inline functions follows the GNU C semantics. If a function definition has **extern inline** explicitly specified, the compiler uses the **extern inline** definition only for inlining. The behavior resembles macro expansion. If an **extern inline** definition of a function exists in a header file, an external definition for the function without **extern** or **inline** must be available from another file. This definition is used for calls to that function from files that do not include the header file.

The following example illustrates the semantics of **extern inline**. When compiled with the GNU semantics, a noninline function body is not generated for `two()`.

```
inline.h:
   #include<stdio.h>

   extern inline void two(void){  // GNU C uses this definition only for inlining
      printf("From inline.h\n");
   }

main.c:
   #include "inline.h"

   int main(void){
      void (*pTwo)() = two;
      two();
      (*pTwo)();
   }

two.c:
   #include<stdio.h>

      void two(){
      printf("In two.c\n");
   }
```

The output below shows the results when the first function call to `two` has indeed been inlined.

```
Using the gcc semantics for the inline keyword:
   From inline.h
   In two.c
```

The compiler might still choose not to inline the **extern inline** function `two`, despite the presence of the **inline** function specifier.

Related References
- "Member Functions" on page 264
- "extern Storage Class Specifier" on page 38

Nested Functions

C A nested function is a function defined inside the definition of another function. It can be defined wherever a variable declaration is permitted, which allows nested functions within nested functions. Within the containing function, the nested function can be declared prior to being defined by using the **auto** keyword. Otherwise, a nested function has internal linkage. The language feature is an orthogonal extension to C89 and C99, implemented to facilitate porting programs developed with GNU C.

A nested function can access all identifiers of the containing function that precede its definition.

Restrictions and limitations

A nested function must not be called after the containing function exits.

A nested function cannot use a **goto** statement to jump to a label in the containing function, or to a local label declared with the **__label__** keyword inherited from the containing function.

Related References

- "Locally Declared Labels" on page 190

Chapter 8. Statements

A statement, the smallest independent computational unit, specifies an action to be performed. In most cases, statements are executed in sequence. The following is a summary of the statements available in C and C++:

- labeled statements
  - identifier labels
  - **case** labels
  - **default** labels
- expression statements
- block or compound statements
- selection statements
  - **if** statements
  - **switch** statements
- iteration statements
  - **while** statements
  - **do** statements
  - **for** statements
- jump statements
  - **break** statements
  - **continue** statements
  - **return** statements
  - **goto** statements
- declaration statements
- ▶ C++ **try** blocks

Labels

There are three kinds of labels: identifier, case, and default.

Identifier label statements have the following form:

```
►►—identifier—:—statement——————————————————————————————————————►◄
```

The label consists of the *identifier* and the colon (**:**) character.

▶ C A label name must be unique within the function in which it appears.

▶ C++ In C++, an identifier label may only be used as the target of a **goto** statement. A **goto** statement can use a label before its definition. Identifier labels have their own name space; you do not have to worry about identifier labels conflicting with other identifiers. However, you may not redeclare a label within a function.

Case and default label statements only appear in **switch** statements. These labels are accessible only within the closest enclosing **switch** statement.

Case statements have the following form:

```
►►—case—constant_expression—:—statement————————————————————————►◄
```

Default label statements have the following form:

```
►►──default──:──statement─────────────────────────────────────────────►◄
```

Examples of Labels

```
comment_complete : ;           /* null statement label */
test_for_null : if (NULL == pointer)
```

Related References

- “goto Statement” on page 206
- “switch Statement” on page 195

Locally Declared Labels

A locally declared label, or *local label*, is an identifier label that is declared at the beginning of a statement expression and for which the scope is the statement expression in which it is declared and defined. This language feature is an orthogonal extension of C and C++ to facilitate handling programs developed with GNU C.

A local label can be used as the target of a **goto** statement, jumping to it from within the same block in which it was declared. This language extension is particularly useful for writing macros that contain nested loops, capitalizing on the difference between its statement scope and the function scope of an ordinary label.

The syntax is as follows:

```
                 ┌─,──────────┐
                 ▼            │
►►──__label__────identifier───┴──;────────────────────────────────────►◄
```

In a statement expression, the declaration of a local label must appear immediately after the left parenthesis and left brace, and must precede any ordinary declarations and statements. The label is defined in the usual way, with a name and a colon, within the statements of the statement expression.

Related References

- “Labels” on page 189

Labels as Values

The address of a label defined in the current function or a containing function can be obtained and used as a value wherever a constant of type **void*** is valid. The address is the return value when the label is the operand of the unary operator **&&**. The ability to use the address of label as a value is an orthogonal extension to C99 and C++, implemented to facilitate porting programs developed with GNU C.

In the following example, the computed goto statements use the values of `label1` and `label2` to jump to those spots in the function.

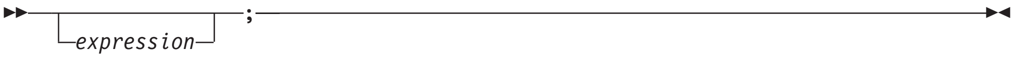
```
int main()
{
   void * ptr1, *ptr2;
   ...
   label1: ...
   ...
   label2: ...
   ...
   ptr1 = &&label1;
   ptr2 = &&label2;
   if (...) {
      goto *ptr1;
   } else {
      goto *ptr2;
   }
   ...
}
```

Related References
- "Label Value Operator &&" on page 126
- "Computed goto" on page 207

Expression Statements

An *expression statement* contains an expression. The expression can be null.

An expression statement has the form:



An expression statement evaluates *expression*, then discards the value of the expression. An expression statement without an expression is a null statement.

Examples of Expressions

```
printf("Account Number: \n");          /* call to the printf    */
marks = dollars * exch_rate;                /* assignment to marks     */
(difference < 0) ? ++losses : ++gain;  /* conditional increment */
```

Related References
- Chapter 5, "Expressions and Operators," on page 99

Resolving Ambiguous Statements in C++

C++ The C++ syntax does not disambiguate between expression statements and declaration statements. The ambiguity arises when an expression statement has a function-style cast as its left-most subexpression. (Note that, because C does not support function-style casts, this ambiguity does not occur in C programs.) If the statement can be interpreted both as a declaration and as an expression, the statement is interpreted as a declaration statement.

Note: The ambiguity is resolved only on a syntactic level. The disambiguation does not use the meaning of the names, except to assess whether or not they are type names.

The following expressions disambiguate into expression statements because the ambiguous subexpression is followed by an assignment or an operator. `type_spec` in the expressions can be any type specifier:

```
type_spec(i)++;             // expression statement
type_spec(i,3)<<d;          // expression statement
type_spec(i)->l=24;         // expression statement
```

In the following examples, the ambiguity cannot be resolved syntactically, and the statements are interpreted as declarations. `type_spec` is any type specifier:

```
type_spec(*i)(int);         // declaration
type_spec(j)[5];            // declaration
type_spec(m) = { 1, 2 };    // declaration
type_spec(*k) (float(3));   // declaration
```

The last statement above causes a compile-time error because you cannot initialize a pointer with a float value.

Any ambiguous statement that is not resolved by the above rules is by default a declaration statement. All of the following are declaration statements:

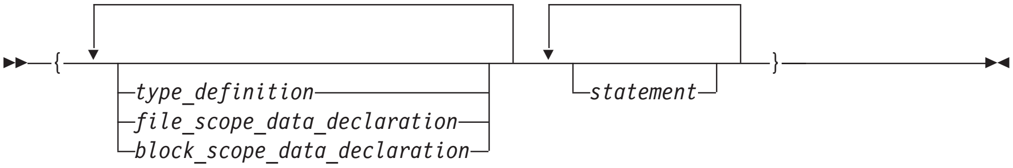
```
type_spec(a);               // declaration
type_spec(*b)();            // declaration
type_spec(c)=23;            // declaration
type_spec(d),e,f,g=0;       // declaration
type_spec(h)(e,3);          // declaration
```

Related References
- Chapter 3, “Declarations,” on page 31
- Chapter 5, “Expressions and Operators,” on page 99
- “Function Call Operator ( )” on page 109

Block Statement

A *block statement*, or *compound statement*, lets you group any number of data definitions, declarations, and statements into one statement. All definitions, declarations, and statements enclosed within a single set of braces are treated as a single statement. You can use a block wherever a single statement is allowed.

A block statement has the form:



C At the C89 language level, definitions and declarations must precede any statements.

For C at the C99 language level and for Standard C++ and C++98, declarations and definitions can appear anywhere, mixed in with other code.

A block defines a local scope. If a data object is usable within a block and its identifier is not redefined, all nested blocks can use that data object.

Example of Blocks

The following program shows how the values of data objects change in nested blocks:

```
/**
 ** This example shows how data objects change in nested blocks.
 **/
  #include <stdio.h>

  int main(void)
  {
      int x = 1;                      /* Initialize x to 1  */
      int y = 3;

      if (y > 0)
      {
         int x = 2;                   /* Initialize x to 2  */
         printf("second x = %4d\n", x);
      }
      printf("first  x = %4d\n", x);

      return(0);
   }
```

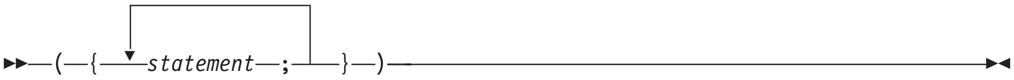
The program produces the following output:

```
second x = 2
first  x =    1
```

Two variables named x are defined in main. The first definition of x retains storage while main is running. However, because the second definition of x occurs within a nested block, printf("second x = %4d\n", x); recognizes x as the variable defined on the previous line. Because printf("first x = %4d\n", x); is not part of the nested block, x is recognized as the first definition of x.

Statement Expressions

C++ A compound statement is a sequence of statements enclosed by braces. In GNU C, a compound statement inside parentheses may appear as an expression in what is called a *statement expression*. This construct, an orthogonal extension of GNU C, is currently supported only in IBM C++.



The value of a statement expression is the value of the last simple expression to appear in the entire construct. If the last statement is not an expression, then the construct is of type **void** and has no value.

The feature can be combined with the **typeof** operator to create complex function-like macros in which each operand is evaluated only once. For example:

```
#define SWAP(a,b) ( {__typeof__(a) temp; temp=a; a=b; b=temp;} )
```

Related References
- "Labels as Values" on page 190
- "typeof Operator" on page 125

if Statement

An **if** statement is a selection statement that allows more than one possible flow of control.

C++ An *if statement* lets you conditionally process a statement when the specified test expression, implicitly converted to **bool**, evaluates to **true**. If the implicit conversion to **bool** fails the program is ill-formed.

C In C, an **if** statement lets you conditionally process a statement when the specified test expression evaluates to a nonzero value. The test expression must be of arithmetic or pointer type.

You can optionally specify an **else** clause on the **if** statement. If the test expression evaluates to **false** (or in C, a zero value) and an **else** clause exists, the statement associated with the **else** clause runs. If the test expression evaluates to **true**, the statement following the expression runs and the **else** clause is ignored.

An **if** statement has the form:

```
►►—if—(—expression—)—statement——┬——————————————————┬——————————————————————►◄
                                └else—statement—┘
```

When **if** statements are nested and **else** clauses are present, a given **else** is associated with the closest preceding **if** statement within the same block.

A single statement following any selection statements (**if**, **switch**) is treated as a compound statement containing the original statement. As a result any variables declared on that statement will be out of scope after the **if** statement. For example:

```
if (x)
int i;
```

is equivalent to:

```
if (x)
{  int i; }
```

Variable i is visible only within the **if** statement. The same rule applies to the **else** part of the **if** statement.

Examples of if Statements

The following example causes grade to receive the value A if the value of score is greater than or equal to 90.

```
if (score >= 90)
   grade = 'A';
```

The following example displays Number is positive if the value of number is greater than or equal to 0. If the value of number is less than 0, it displays Number is negative.

```
if (number >= 0)
   printf("Number is positive\n");
else
   printf("Number is negative\n");
```

The following example shows a nested **if** statement:

```
if (paygrade == 7)
   if (level >= 0 && level <= 8)
      salary *= 1.05;
   else
      salary *= 1.04;
else
   salary *= 1.06;
cout << "salary is " << salary << endl;
```

The following example shows a nested **if** statement that does not have an **else** clause. Because an **else** clause always associates with the closest **if** statement, braces might be needed to force a particular **else** clause to associate with the correct **if** statement. In this example, omitting the braces would cause the **else** clause to associate with the nested **if** statement.

```
if (kegs > 0) {
   if (furlongs > kegs)
      fpk = furlongs/kegs;
}
else
   fpk = 0;
```

The following example shows an **if** statement nested within an **else** clause. This example tests multiple conditions. The tests are made in order of their appearance. If one test evaluates to a nonzero value, a statement runs and the entire **if** statement ends.

```
if (value > 0)
   ++increase;
else if (value == 0)
   ++break_even;
else
   ++decrease;
```

switch Statement

A *switch statement* is a selection statement that lets you transfer control to different statements within the **switch** body depending on the value of the switch expression. The **switch** expression must evaluate to an integral or enumeration value. The body of the **switch** statement contains *case clauses* that consist of

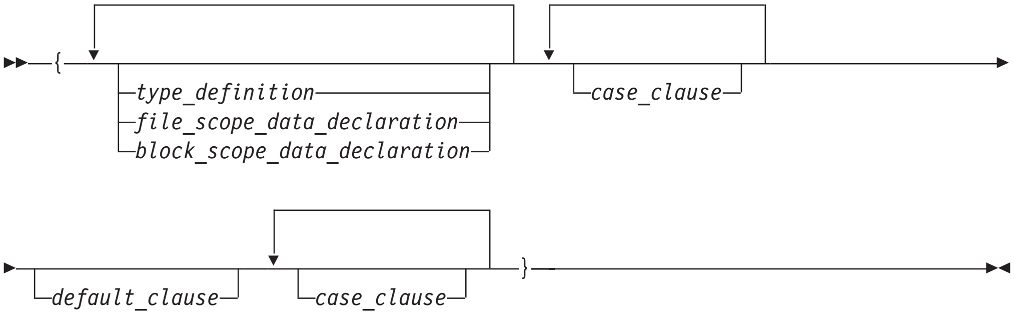
- A **case** label
- An optional **default** label
- A **case** expression
- A list of statements.

If the value of the **switch** expression equals the value of one of the case expressions, the statements following that case expression are processed. If not, the default label statements, if any, are processed.

A **switch** statement has the form:

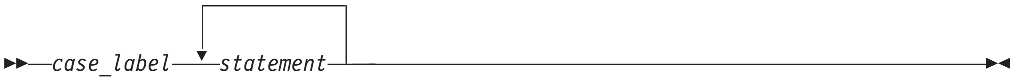
```
►►—switch—(—expression—)—switch_body——————————————————————►◄
```

The *switch body* is enclosed in braces and can contain definitions, declarations, *case clauses*, and a *default clause*. Each case clause and default clause can contain statements.

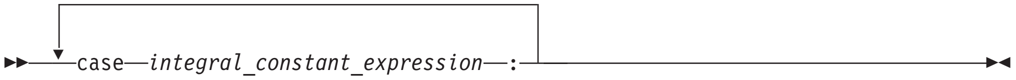


Note: An initializer within a *type_definition*, *file_scope_data_declaration* or *block_scope_data_declaration* is ignored.

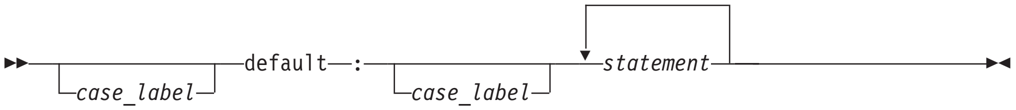
A *case clause* contains a *case label* followed by any number of statements. A case clause has the form:



A *case label* contains the word **case** followed by an integral constant expression and a colon. The value of each integral constant expression must represent a different value; you cannot have duplicate **case** labels. Anywhere you can put one **case** label, you can put multiple **case** labels. A case label has the form:



A *default clause* contains a **default** label followed by one or more statements. You can put a **case** label on either side of the **default** label. A **switch** statement can have only one **default** label. A *default_clause* has the form:



The **switch** statement passes control to the statement following one of the labels or to the statement following the **switch** body. The value of the expression that precedes the **switch** body determines which statement receives control. This expression is called the *switch expression*.

The value of the **switch** expression is compared with the value of the expression in each **case** label. If a matching value is found, control is passed to the statement following the **case** label that contains the matching value. If there is no matching value but there is a **default** label in the **switch** body, control passes to the **default** labelled statement. If no matching value is found, and there is no **default** label anywhere in the **switch** body, no part of the **switch** body is processed.

When control passes to a statement in the **switch** body, control only leaves the **switch** body when a **break** statement is encountered or the last statement in the **switch** body is processed.

If necessary, an integral promotion is performed on the controlling expression, and all expressions in the **case** statements are converted to the same type as the controlling expression. The **switch** expression can also be of class type if there is a single conversion to integral or enumeration type.

Restrictions and Limitations

You can put data definitions at the beginning of the **switch** body, but the compiler does not initialize **auto** and **register** variables at the beginning of a **switch** body. You can have declarations in the body of the **switch** statement.

You cannot use a **switch** statement to jump over initializations.

C When the scope of an identifier with a variably modified type includes a case or default label of a switch statement, the entire switch statement is considered to be within the scope of that identifier. That is, the declaration of the identifier must precede the switch statement.

C++ In C++, you cannot transfer control over a declaration containing an explicit or implicit initializer unless the declaration is located in an inner block that is completely bypassed by the transfer of control. All declarations within the body of a **switch** statement that contain initializers must be contained in an inner block.

Examples of switch Statements

The following **switch** statement contains several **case** clauses and one **default** clause. Each clause contains a function call and a **break** statement. The **break** statements prevent control from passing down through each statement in the **switch** body.

If the **switch** expression evaluated to `'/'`, the switch statement would call the function `divide`. Control would then pass to the statement following the **switch** body.

```
char key;

printf("Enter an arithmetic operator\n");
scanf("%c",&key);

switch (key)
{
   case '+':
      add();
      break;

   case '-':
      subtract();
      break;

   case '*':
      multiply();
      break;

   case '/':
      divide();
      break;

   default:
      printf("invalid key\n");
      break;
}
```

If the switch expression matches a case expression, the statements following the case expression are processed until a **break** statement is encountered or the end of the **switch** body is reached. In the following example, **break** statements are not present. If the value of `text[i]` is equal to `'A'`, all three counters are incremented. If the value of `text[i]` is equal to `'a'`, `lettera` and `total` are increased. Only `total` is increased if `text[i]` is not equal to `'A'` or `'a'`.

```
char text[100];
int capa, lettera, total;

// ...

for (i=0; i<sizeof(text); i++) {
    switch (text[i])
    {
       case 'A':
         capa++;
       case 'a':
         lettera++;
       default:
         total++;
    }
}
```

The following **switch** statement performs the same statements for more than one **case** label:

```
/**
 ** This example contains a switch statement that performs
 ** the same statement for more than one case label.
 **/

#include <stdio.h>

int main(void)
{
  int month;

  /* Read in a month value */
  printf("Enter month: ");
  scanf("%d", &month);

  /* Tell what season it falls into */
  switch (month)
  {
     case 12:
     case 1:
     case 2:
        printf("month %d is a winter month\n", month);
        break;

     case 3:
     case 4:
     case 5:
        printf("month %d is a spring month\n", month);
        break;

     case 6:
     case 7:
     case 8:
        printf("month %d is a summer month\n", month);
        break;

     case 9:
     case 10:
     case 11:
        printf("month %d is a fall month\n", month);
        break;

     case 66:
     case 99:
     default:
        printf("month %d is not a valid month\n", month);
  }

  return(0);
}
```

If the expression month has the value 3, control passes to the statement:

```
printf("month %d is a spring month\n",
month);
```

The **break** statement passes control to the statement following the **switch** body.

while Statement

A *while statement* repeatedly runs the body of a loop until the controlling expression evaluates to **false** (or `0` in C).

A **while** statement has the form:

```
►►──while──(──expression──)──statement────────────────────────────────►◄
```

C The *expression* must be of arithmetic or pointer type.

The expression is evaluated to determine whether or not to process the body of the loop.

C++ The expression must be convertible to **bool**.

If the expression evaluates to **false**, the body of the loop never runs. If the expression does not evaluate to **false**, the loop body is processed. After the body has run, control passes back to the expression. Further processing depends on the value of the condition.

A **break**, **return**, or **goto** statement can cause a **while** statement to end, even when the condition does not evaluate to **false**.

C++ A **throw** expression also can cause a **while** statement to end prior to the condition being evaluated.

Example of while Statements

In the following program, `item[index]` triples and is printed out, as long as the value of the expression `++index` is less than `MAX_INDEX`. When `++index` evaluates to `MAX_INDEX`, the **while** statement ends.

```
/**
 ** This example illustrates the while statement.
 **/

#define MAX_INDEX  (sizeof(item) / sizeof(item[0]))
#include <stdio.h>

int main(void)
{
   static int item[ ] = { 12, 55, 62, 85, 102 };
   int index = 0;

   while (index < MAX_INDEX)
   {
      item[index] *= 3;
      printf("item[%d] = %d\n", index, item[index]);
      ++index;
   }

   return(0);
}
```

do Statement

A *do statement* repeatedly runs a statement until the test expression evaluates to **false** (or `0` in C). Because of the order of processing, the statement is run at least once.

A **do** statement has the form:

```
►►—do—statement—while—(—expression—)—;————————————————————————————►◄
```

C++ The controlling *expression* must convertible to type **bool**.

C The *expression* must be of arithmetic or pointer type.

The body of the loop is run before the controlling **while** clause is evaluated. Further processing of the **do** statement depends on the value of the **while** clause. If the **while** clause does not evaluate to **false**, the statement runs again. When the **while** clause evaluates to **false**, the statement ends.

A **break**, **return**, or **goto** statement can cause the processing of a **do** statement to end, even when the **while** clause does not evaluate to **false**.

C++ A **throw** expression also can cause a **while** statement to end prior to the condition being evaluated.

Example of do Statements

The following example keeps incrementing `i` while `i` is less than `5`:

```
#include <stdio.h>

int main(void) {
  int i = 0;
  do {
    i++;
    printf("Value of i: %d\n", i);
  }
  while (i < 5);
  return 0;
}
```

The following is the output of the above example:

```
Value of i: 1
Value of i: 2
Value of i: 3
Value of i: 4
Value of i: 5
```

for Statement

A *for statement* lets you do the following:

- Evaluate an expression before the first iteration of the statement (*initialization*)

- Specify an expression to determine whether or not the statement should be processed (the *condition*)
- Evaluate an expression after each iteration of the statement (often used to increment for each iteration)
- Repeatedly process the statement if the controlling part does not evaluate to **false** (or `0` in C).

A **for** statement has the form:

```
►►──for──(─┬─────────────┬─;─┬─────────────┬─;─┬─────────────┬─)──────────►
           └─expression1─┘   └─expression2─┘   └─expression3─┘

►─statement───────────────────────────────────────────────────────────────►◄
```

expression1 Is the *initialization expression*. It is evaluated only before the *statement* is processed for the first time. You can use this expression to initialize a variable. If you do not want to evaluate an expression prior to the first iteration of the statement, you can omit this expression.

expression2 Is the *conditional expression*. It is evaluated before each iteration of the *statement*.

C It must evaluate to an arithmetic or pointer type.

If it evaluates to **false** (or `0` in C), the statement is not processed and control moves to the next statement following the **for** statement. If *expression2* does not evaluate to **false**, the statement is processed. If you omit *expression2*, it is as if the expression had been replaced by **true**, and the **for** statement is not terminated by failure of this condition.

expression3 Is evaluated after each iteration of the *statement*. This expression is often used for incrementing, decrementing, or assigning to a variable. This expression is optional.

A **break**, **return**, or **goto** statement can cause a **for** statement to end, even when the second expression does not evaluate to **false**. If you omit *expression2*, you must use a **break**, **return**, or **goto** statement to end the **for** statement.

You can also use *expression1* to declare a variable as well as initialize it. If you declare a variable in this expression, or anywhere else in *statement*, that variable goes out of scope at the end of the **for** loop.

C++ You can set a compiler option that allows a variable declared in the scope of a **for** statement to have a scope that is not local to the **for** statement.

Examples of for Statements

The following **for** statement prints the value of `count` `20` times. The **for** statement initially sets the value of `count` to `1`. After each iteration of the statement, `count` is incremented.

```
int count;
for (count = 1; count <= 20; count++)
   printf("count = %d\n", count);
```

The following sequence of statements accomplishes the same task. Note the use of the **while** statement instead of the **for** statement.

```
int count = 1;
while (count <= 20)
{
   printf("count = %d\n", count);
   count++;
}
```

The following **for** statement does not contain an initialization expression:

```
for (; index > 10; --index)
{
   list[index] = var1 + var2;
   printf("list[%d] = %d\n", index,
   list[index]);
}
```

The following **for** statement will continue running until `scanf` receives the letter `e`:

```
for (;;)
{
   scanf("%c", &letter);
   if (letter == '\n')
      continue;
   if (letter == 'e')
      break;
   printf("You entered the letter %c\n", letter);
}
```

The following **for** statement contains multiple initializations and increments. The comma operator makes this construction possible. The first comma in the **for** expression is a punctuator for a declaration. It declares and initializes two integers, `i` and `j`. The second comma, a comma operator, allows both `i` and `j` to be incremented at each step through the loop.

```
for (int i = 0,
j = 50; i < 10; ++i, j += 50)
{
   cout << "i = " << i << "and j = " << j
   << endl;
}
```

The following example shows a nested **for** statement. It prints the values of an array having the dimensions `[5][3]`.

```
for (row = 0; row < 5; row++)
   for (column = 0; column < 3; column++)
      printf("%d\n",
      table[row][column]);
```

The outer statement is processed as long as the value of `row` is less than `5`. Each time the outer **for** statement is executed, the inner **for** statement sets the initial value of `column` to zero and the statement of the inner **for** statement is executed `3` times. The inner statement is executed as long as the value of `column` is less than `3`.

break Statement

A *break statement* lets you end an *iterative* (**do**, **for**, or **while**) statement or a **switch** statement and exit from it at any point other than the logical end. A **break** may only appear on one of these statements.

A **break** statement has the form:

```
►►──break──;──────────────────────────────────────────────────────────►◄
```

In an iterative statement, the **break** statement ends the loop and moves control to the next statement outside the loop. Within nested statements, the **break** statement ends only the smallest enclosing **do**, **for**, **switch**, or **while** statement.

In a **switch** statement, the **break** passes control out of the **switch** body to the next statement outside the **switch** statement.

continue Statement

A *continue statement* ends the current iteration of a loop. Program control is passed from the **continue** statement to the end of the loop body.

A **continue** statement has the form:

```
►►──continue──;───────────────────────────────────────────────────────►◄
```

A **continue** statement can only appear within the body of an iterative statement, such as **do**, **for**, or **while**.

The **continue** statement ends the processing of the action part of an iterative statement and moves control to the loop continuation portion of the statement. For example, if the iterative statement is a **for** statement, control moves to the third expression in the condition part of the statement, then to the second expression (the test) in the condition part of the statement.

Within nested statements, the **continue** statement ends only the current iteration of the **do**, **for**, or **while** statement immediately enclosing it.

Examples of continue Statements

The following example shows a **continue** statement in a **for** statement. The **continue** statement causes processing to skip over those elements of the array `rates` that have values less than or equal to 1.

```
/**
 ** This example shows a continue statement in a for statement.
 **/

#include <stdio.h>
#define  SIZE  5

int main(void)
{
   int i;
   static float rates[SIZE] = { 1.45, 0.05, 1.88, 2.00, 0.75 };

   printf("Rates over 1.00\n");
   for (i = 0; i < SIZE; i++)
   {
      if (rates[i] <= 1.00)  /*  skip rates <= 1.00  */
         continue;
      printf("rate = %.2f\n", rates[i]);
   }

   return(0);
}
```

The program produces the following output:

```
Rates over 1.00
rate = 1.45
rate = 1.88
rate = 2.00
```

The following example shows a **continue** statement in a nested loop. When the inner loop encounters a number in the array `strings`, that iteration of the loop ends. Processing continues with the third expression of the inner loop. The inner loop ends when the '\0' escape sequence is encountered.

```
/**
 ** This program counts the characters in strings that are part
 ** of an array of pointers to characters.  The count excludes
 ** the digits 0 through 9.
 **/

#include <stdio.h>
#define  SIZE  3

int main(void)
{
   static char *strings[SIZE] = { "ab", "c5d", "e5" };
   int i;
   int letter_count = 0;
   char *pointer;
   for (i = 0; i < SIZE; i++)             /* for each string   */
                                           /* for each each character */
      for (pointer = strings[i]; *pointer != '\0';
      ++pointer)
      {                                  /* if a number      */
         if (*pointer >= '0' && *pointer <= '9')
            continue;
         letter_count++;
      }
   printf("letter count = %d\n", letter_count);

   return(0);
}
```

The program produces the following output:

```
letter count = 5
```

return Statement

A *return statement* ends the processing of the current function and returns control to the caller of the function.

A **return** statement has one of two forms:

```
►►──return──┬────────────┬──;──────────────────────────────────────────►◄
            └─expression─┘
```

A value-returning function must include an *expression* in the **return** statement. A function with a return type is `void` cannot contain an *expression* in its return statement.

For a function of return type `void`, a return statement is not strictly necessary. If the end of such a function is reached without encountering a **return** statement, control is passed to the caller as if a **return** statement without an expression were encountered. In other words, an implicit return takes place upon completion of the final statement, and control automatically returns to the calling function. A function can contain multiple **return** statements. For example:

```
void copy( int *a, int *b, int c)
{
   /* Copy array a into b, assuming both arrays are the same size */

   if (!a || !b)       /* if either pointer is 0, return */
      return;

   if (a == b)         /* if both parameters refer */
      return;          /*    to same array, return */

   if (c == 0)         /* nothing to copy */
      return;

   for (int i = 0; i < c; ++i;) /* do the copying */
      b[i] = a[1];
                       /* implicit return */
}
```

In this example, the **return** statement is used to cause a premature termination of the function, similar to a **break** statement.

An expression appearing in a **return** statement is converted to the return type of the function in which the statement appears. If no implicit conversion is possible, the **return** statement is invalid.

Value of a return Expression and Function Value

If an expression is present on a **return** statement, the value of the expression is returned to the caller. If the data type of the expression is different from the function return type, conversion of the return value takes place as if the value of the expression were assigned to an object with the same function return type.

The value of the **return** statement for a function of return type `void` means that the function does not return a value. If an expression is not given on a **return** statement in a function declared with a non-`void` return type, the complier issues an error message.

You cannot use a **return** statement with an expression when the function is declared as returning type **void**.

Examples of return Statements

```
return;            /* Returns no value            */
return result;     /* Returns the value of result */
return 1;          /* Returns the value 1         */
return (x * x);    /* Returns the value of x * x  */
```

The following function searches through an array of integers to determine if a match exists for the variable `number`. If a match exists, the function `match` returns the value of `i`. If a match does not exist, the function `match` returns the value `-1` (negative one).

```
int match(int number, int array[ ], int n)
{
   int i;

   for (i = 0; i < n; i++)
```

```
      if (number == array[i])
         return (i);
   return(-1);
}
```

goto Statement

A *goto statement* causes your program to unconditionally transfer control to the statement associated with the label specified on the **goto** statement.

A **goto** statement has the form:

```
▶▶—goto—label_identifier—;——————————————————————————————————▶◀
```

Because the **goto** statement can interfere with the normal sequence of processing, it makes a program more difficult to read and maintain. Often, a **break** statement, a **continue** statement, or a function call can eliminate the need for a **goto** statement.

If an active block is exited using a **goto** statement, any local variables are destroyed when control is transferred from that block.

You cannot use a **goto** statement to jump over initializations.

▶ C A **goto** statement is allowed to jump within the scope of a variable length array, but not past any declarations of objects with variably modified types.

Example of goto Statements

The following example shows a **goto** statement that is used to jump out of a nested loop. This function could be written without using a **goto** statement.

```
/**
 ** This example shows a goto statement that is used to
 ** jump out of a nested loop.
 **/

#include <stdio.h>
void display(int matrix[3][3]);

int main(void)
{
   int matrix[3][3]=    {1,2,3,4,5,2,8,9,10};
   display(matrix);
   return(0);
}

void display(int matrix[3][3])
{
   int i, j;

   for (i = 0; i < 3; i++)
      for (j = 0; j < 3; j++)
      {
         if ( (matrix[i][j] < 1) || (matrix[i][j] > 6) )
            goto out_of_bounds;
         printf("matrix[%d][%d] = %d\n", i, j, matrix[i][j]);
      }
   return;
   out_of_bounds: printf("number must be 1 through 6\n");
}
```

Computed goto

A computed goto is a goto statement for which the target is a label from the same function. The address of the label is a constant of type **void***, and is obtained by applying the unary label value operator **&&** to the label. The target of a computed goto is known at run time, and all computed goto statements from the same function will have the same targets. The language feature is an orthogonal extension to C99 and C++, implemented to facilitate porting programs developed with GNU C.

A computed goto is of the form

```
►►──goto──*expression──;─────────────────────────────────────────────►◄
```

where **expression* is an expression of type **void***.

Related References

- “Labels as Values” on page 190
- “Label Value Operator &&” on page 126

Null Statement

The *null statement* performs no operation. It has the form:

```
►►──;────────────────────────────────────────────────────────────────►◄
```

A `null` statement can hold the label of a labeled statement or complete the syntax of an iterative statement.

Examples of Null Statements

The following example initializes the elements of the array `price`. Because the initializations occur within the **for** expressions, a statement is only needed to finish the **for** syntax; no operations are required.

```
for (i = 0; i < 3; price[i++] = 0)
   ;
```

A null statement can be used when a label is needed before the end of a block statement. For example:

```
void func(void) {
  if (error_detected)
    goto depart;
  /* further processing */
  depart: ;  /* null statement required */
}
```

Chapter 9. Preprocessor Directives

The preprocessor is a program that is invoked by the compiler to process code before compilation. Commands for that program, known as *directives*, are lines of the source file beginning with the character #, which distinguishes them from lines of source program text. The effect of each preprocessor directive is a change to the text of the source code, and the result is a new source code file, which does not contain the directives. The preprocessed source code, an intermediate file, must be a valid C or C++ program, because it becomes the input to the compiler.

The syntax of preprocessor directives is independent of, but similar to, the syntax of the rest of the language, and the lexical conventions of the preprocessor differ from those of the compiler. The preprocessor recognizes the normal C and C++ tokens, as well as other characters that enable the preprocessor to recognize file names, the presence and absence of white space, and the location of end-of-line markers.

Preprocessor directives and the related subject of macro expansion are discussed in this section. After an overview of preprocessor directives, the topics covered include textual macros, file inclusion, ISO standard and predefined macro names, conditional compilation directives, and pragmas.

Preprocessor Overview

Preprocessing is a preliminary operation on C and C++ files before they are passed to the compiler. It allows you to do the following:

- Replace tokens in the current file with specified replacement tokens
- Imbed files within the current file
- Conditionally compile sections of the current file
- Generate diagnostic messages
- Change the line number of the next line of source and change the file name of the current file
- Apply machine-specific rules to specified sections of code

A *token* is a series of characters delimited by white space. The only white space allowed on a preprocessor directive is the space, horizontal tab, vertical tab, form feed, and comments. The new-line character can also separate preprocessor tokens.

The preprocessed source program file must be a valid C or C++ program.

The preprocessor is controlled by the following directives:

`#define` Defines a macro.

`#undef` Removes a preprocessor macro definition.

`#error` Defines text for a compile-time error message.

`#include` Inserts text from another source file.

`#if` Conditionally suppresses portions of source code, depending on the result of a constant expression.

`#ifdef` Conditionally includes source text if a macro name is defined.

`#ifndef` Conditionally includes source text if a macro name is not defined.

`#else` Conditionally includes source text if the previous **#if**, **#ifdef**, **#ifndef**, or **#elif** test fails.

`#elif` Conditionally includes source text if the previous **#if**, **#ifdef**, **#ifndef**, or **#elif** test fails, depending on the value of a constant expression.

`#endif` Ends conditional text.

`#line` Supplies a line number for compiler messages.

`#pragma` Specifies implementation-defined instructions to the compiler.

Preprocessor Directive Format

Preprocessor directives begin with the # token followed by a preprocessor keyword. The # token must appear as the first character that is not white space on a line. The # is not part of the directive name and can be separated from the name with white spaces.

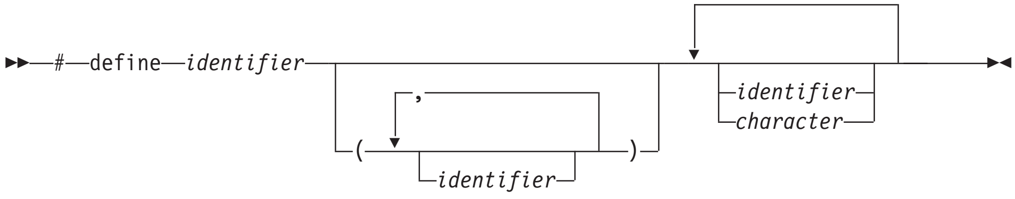
A preprocessor directive ends at the new-line character unless the last character of the line is the \ (backslash) character. If the \ character appears as the last character in the preprocessor line, the preprocessor interprets the \ and the new-line character as a continuation marker. The preprocessor deletes the \ (and the following new-line character) and splices the physical source lines into continuous logical lines. White space is allowed between backslash and the end of line character or the physical end of record. However,this white space is usually not visible during editing.

Except for some **#pragma** directives, preprocessor directives can appear anywhere in a program.

Macro Definition and Expansion (#define)

A *preprocessor define directive* directs the preprocessor to replace all subsequent occurrences of a macro with specified replacement tokens.

A preprocessor **#define** directive has the form:



The **#define** directive can contain an object-like definition or a function-like definition.

#define versus `const`

- The **#define** directive can be used to create a name for a numerical, character, or string constant, whereas a `const` object of any type can be declared.
- A `const` object is subject to the scoping rules for variables, whereas a constant created using **#define** is not.
- Unlike a `const` object, the value of a macro does not appear in the intermediate source code used by the compiler because they are expanded inline. The inline expansion makes the macro value unavailable to the debugger.
- A macro can be used in a constant expression, such as an array bound, whereas a `const` object cannot.

- C++ The compiler does not type-check a macro, including macro arguments.

Related References
- “Object-Like Macros”
- “Function-Like Macros”
- “The const Type Qualifier” on page 73

Object-Like Macros

An *object-like macro definition* replaces a single identifier with the specified replacement tokens. The following object-like definition causes the preprocessor to replace all subsequent instances of the identifier `COUNT` with the constant `1000` :

```
#define COUNT 1000
```

If the statement

```
int arry[COUNT];
```

appears after this definition and in the same file as the definition, the preprocessor would change the statement to

```
int arry[1000];
```

in the output of the preprocessor.

Other definitions can make reference to the identifier `COUNT`:

```
#define MAX_COUNT COUNT + 100
```

The preprocessor replaces each subsequent occurrence of `MAX_COUNT` with `COUNT + 100`, which the preprocessor then replaces with `1000 + 100`.

If a number that is partially built by a macro expansion is produced, the preprocessor does not consider the result to be a single value. For example, the following will not result in the value 10.2 but in a syntax error.

```
#define a 10
a.2
```

Identifiers that are partially built from a macro expansion may not be produced. Therefore, the following example contains two identifiers and results in a syntax error:

```
#define d efg
abcd
```

Function-Like Macros

More complex than object-like macros, a function-like macro definition declares the names of formal parameters within parentheses, separated by commas. An empty formal parameter list is legal: such a macro can be used to simulate a function that takes no arguments. C adds support for function-like macros with a variable number of arguments.

C++ C++ supports function-like macros with a variable number of arguments, as a language extension for compatibility with C.

Function-like macro definition:
An identifier followed by a parameter list in parentheses and the replacement tokens. The parameters are imbedded in the replacement code.

White space cannot separate the identifier (which is the name of the macro) and the left parenthesis of the parameter list. A comma must separate each parameter.

For portability, you should not have more than 31 parameters for a macro. The parameter list may end with an ellipsis (...). In this case, the identifier `__VA_ARGS__` may appear in the replacement list.

Function-like macro invocation:

An identifier followed by a comma-separated list of arguments in parentheses. The number of arguments should match the number of parameters in the macro definition, unless the parameter list in the definition ends with an ellipsis. In this latter case, the number of arguments in the invocation should exceed the number of parameters in the definition. The excess are called *trailing arguments*. Once the preprocessor identifies a function-like macro invocation, argument substitution takes place. A parameter in the replacement code is replaced by the corresponding argument. If trailing arguments are permitted by the macro definition, they are merged with the intervening commas to replace the identifier `__VA_ARGS__`, as if they were a single argument. Any macro invocations contained in the argument itself are completely replaced before the argument replaces its corresponding parameter in the replacement code.

A macro argument can be empty (consisting of zero preprocessing tokens). For example,

```
#define SUM(a,b,c) a + b + c
SUM(1,,3)  /* No error message.
              1 is substituted for a, 3 is substituted for c. */
```

This language feature is an orthogonal extension of C++.

If the identifier list does not end with an ellipsis, the number of arguments in a macro invocation must be the same as the number of parameters in the corresponding macro definition. During parameter substitution, any arguments remaining after all specified arguments have been substituted (including any separating commas) are combined into one argument called the variable argument. The variable argument will replace any occurrence of the identifier `__VA_ARGS__` in the replacement list. The following example illustrates this:

```
#define debug(...)   fprintf(stderr, __VA_ARGS__)

debug("flag");     /*   Becomes fprintf(stderr, "flag");   */
```

Commas in the macro invocation argument list do not act as argument separators when they are:

- In character constants
- In string literals
- Surrounded by parentheses

The following line defines the macro `SUM` as having two parameters `a` and `b` and the replacement tokens `(a + b)`:

```
#define SUM(a,b) (a + b)
```

This definition would cause the preprocessor to change the following statements (if the statements appear after the previous definition):

```
c = SUM(x,y);
c = d * SUM(x,y);
```

In the output of the preprocessor, these statements would appear as:

```
c = (x + y);
c = d * (x + y);
```

Use parentheses to ensure correct evaluation of replacement text. For example, the definition:

```
#define SQR(c)  ((c) * (c))
```

requires parentheses around each parameter `c` in the definition in order to correctly evaluate an expression like:

```
y = SQR(a + b);
```

The preprocessor expands this statement to:

```
y = ((a + b) * (a + b));
```

Without parentheses in the definition, the correct order of evaluation is not preserved, and the preprocessor output is:

```
y = (a + b * a + b);
```

Arguments of the # and ## operators are converted *before* replacement of parameters in a function-like macro.

Once defined, a preprocessor identifier remains defined and in scope independent of the scoping rules of the language. The scope of a macro definition begins at the definition and does not end until a corresponding **#undef** directive is encountered. If there is no corresponding **#undef** directive, the scope of the macro definition lasts until the end of the translation unit.

A recursive macro is not fully expanded. For example, the definition

```
   #define x(a,b) x(a+1,b+1) + 4
```

expands

```
   x(20,10)
```

to

```
   x(20+1,10+1) + 4
```

rather than trying to expand the macro `x` over and over within itself. After the macro `x` is expanded, it is a call to function `x()`.

A definition is not required to specify replacement tokens. The following definition removes all instances of the token `debug` from subsequent lines in the current file:

```
#define debug
```

You can change the definition of a defined identifier or macro with a second preprocessor **#define** directive only if the second preprocessor **#define** directive is preceded by a preprocessor **#undef** directive. The **#undef** directive nullifies the first definition so that the same identifier can be used in a redefinition.

Within the text of the program, the preprocessor does not scan character constants or string constants for macro invocations.

Example of #define Directives

The following program contains two macro definitions and a macro invocation that refers to both of the defined macros:

```
/**
 ** This example illustrates #define directives.
 **/

#include <stdio.h>

#define SQR(s)  ((s) * (s))
#define PRNT(a,b) \
  printf("value 1 = %d\n", a); \
  printf("value 2 = %d\n", b) ;

int main(void)
{
  int x = 2;
  int y = 3;

     PRNT(SQR(x),y);

  return(0);
}
```

After being interpreted by the preprocessor, this program is replaced by code equivalent to the following:

```
#include <stdio.h>

int main(void)
{
  int x = 2;
  int y = 3;

     printf("value 1 = %d\n", ( (x) * (x) ) );
     printf("value 2 = %d\n", y);

  return(0);
}
```

This program produces the following output:

```
value 1 = 4
value 2 = 3
```

Variadic Macro Extensions

Variadic macro extensions refer to two extensions to C99 related to macros with variable number of arguments. One extension is a mechanism for renaming the variable argument identifier from __VA_ARGS__ to a user-defined identifier. This extension is orthogonal to C99. The other extension provides a way to remove the dangling comma in a variadic macro when no variable arguments are specified. This extension is non-orthogonal. Both extensions have been implemented to facilitate porting programs developed with GNU C and C++.

▶ C++ This implementation of C++ extends Standard C++ and C++98 to support the variadic macro extensions for compatibility with C.

An Identifier Instead of __VA_ARGS__

The following examples demonstrate the use of an identifier in place of __VA_ARGS__. The first definition of the macro debug exemplifies the usual usage of __VA_ARGS__. The second definition shows the use of the identifier args in place of __VA_ARGS__.

```
#define debug1(format, ...)  printf(format, __VA_ARGS__)
#define debug2(format, args ...)  printf(format, args)
```

Invocation	Result of Macro Expansion
`debug1("Hello %s\n","World");`	`printf("Hello %s\n","World");`
`debug2("Hello %s\n","World");`	`printf("Hello %s\n","World");`

Trailing Comma Removal

The preprocessor removes the trailing comma if the variable arguments to a function macro are omitted or empty and the comma followed by ## precedes the variable argument identifier in the function macro definition.

Scope of Macro Names (#undef)

A *preprocessor undef directive* causes the preprocessor to end the scope of a preprocessor definition.

A preprocessor **#undef** directive has the form:

```
►►──#──undef──identifier──────────────────────────────────────────►◄
```

If the identifier is not currently defined as a macro, **#undef** is ignored.

Example of #undef Directives

The following directives define BUFFER and SQR:

```
#define BUFFER 512
#define SQR(x) ((x) * (x))
```

The following directives nullify these definitions:

```
#undef BUFFER
#undef SQR
```

Any occurrences of the identifiers BUFFER and SQR that follow these **#undef** directives are not replaced with any replacement tokens. Once the definition of a macro has been removed by an **#undef** directive, the identifier can be used in a new **#define** directive.

# Operator

The # (single number sign) operator converts a parameter of a function-like macro into a character string literal. For example, if macro ABC is defined using the following directive:

```
   #define ABC(x)   #x
```

all subsequent invocations of the macro ABC would be expanded into a character string literal containing the argument passed to ABC. For example:

Invocation	Result of Macro Expansion
`ABC(1)`	"1"
`ABC(Hello there)`	"Hello there"

The # operator should not be confused with the null directive.

Use the # operator in a function-like macro definition according to the following rules:

- A parameter following # operator in a function- like macro is converted into a character string literal containing the argument passed to the macro.
- White-space characters that appear before or after the argument passed to the macro are deleted.
- Multiple white-space characters imbedded within the argument passed to the macro are replaced by a single space character.
- If the argument passed to the macro contains a string literal and if a \ (backslash) character appears within the literal, a second \ character is inserted before the original \ when the macro is expanded.
- If the argument passed to the macro contains a " (double quotation mark) character, a \ character is inserted before the " when the macro is expanded.
- The conversion of an argument into a string literal occurs before macro expansion on that argument.
- If more than one ## operator or # operator appears in the replacement list of a macro definition, the order of evaluation of the operators is not defined.
- If the result of the macro expansion is not a valid character string literal, the behavior is undefined.

Example of the # Operator

The following examples demonstrate the use of the # operator:

```
#define STR(x)       #x
#define XSTR(x)      STR(x)
#define ONE          1
```

Invocation	Result of Macro Expansion
STR(\n "\n" '\n')	"\n \"\\n\" '\\n'"
STR(ONE)	"ONE"
XSTR(ONE)	"1"
XSTR("hello")	"\"hello\""

Macro Concatenation with the ## Operator

The ## (double number sign) operator concatenates two tokens in a macro invocation (text and/or arguments) given in a macro definition.

If a macro XY was defined using the following directive:

```
#define XY(x,y)    x##y
```

the last token of the argument for x is concatenated with the first token of the argument for y.

Use the ## operator according to the following rules:

- The ## operator cannot be the very first or very last item in the replacement list of a macro definition.
- The last token of the item in front of the ## operator is concatenated with first token of the item following the ## operator.
- Concatenation takes place before any macros in arguments are expanded.
- If the result of a concatenation is a valid macro name, it is available for further replacement even if it appears in a context in which it would not normally be available.

- If more than one ## operator and/or # operator appears in the replacement list of a macro definition, the order of evaluation of the operators is not defined.

Examples of the ## Operator

The following examples demonstrate the use of the ## operator:

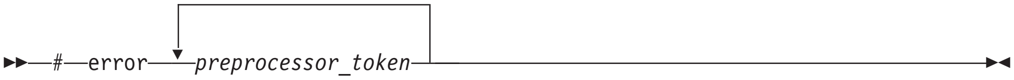
```
#define ArgArg(x, y)          x##y
#define ArgText(x)            x##TEXT
#define TextArg(x)            TEXT##x
#define TextText              TEXT##text
#define Jitter                1
#define bug                   2
#define Jitterbug             3
```

Invocation	Result of Macro Expansion
ArgArg(lady, bug)	"ladybug"
ArgText(con)	"conTEXT"
TextArg(book)	"TEXTbook"
TextText	"TEXTtext"
ArgArg(Jitter, bug)	3

Preprocessor Error Directive (#error)

A *preprocessor error directive* causes the preprocessor to generate an error message and causes the compilation to fail.

A **#error** directive has the form:



The argument *preprocessor_token* is not subject to macro expansion.

The **#error** directive is often used in the **#else** portion of a **#if–#elif–#else** construct, as a safety check during compilation. For example, **#error** directives in the source file can prevent code generation if a section of the program is reached that should be bypassed.

For example, the directive

```
#define BUFFER_SIZE 255

#if BUFFER_SIZE < 256
#error "BUFFER_SIZE is too small."
#endif
```

generates the error message:

```
BUFFER_SIZE is too small.
```

Preprocessor Warning Directive (#warning)

A *preprocessor warning directive* causes the preprocessor to generate a warning message but allows compilation to continue. The argument to **#warning** is not subject to macro expansion.

A **#warning** directive has the form:

```
              ┌────────────────────┐
              ▼                    │
►►──#──warning───preprocessor_token─┴──────────────────────────────────►◄
```

The preprocessor **#warning** directive is an orthogonal language extension provided to facilitate handling programs developed with GNU C. The IBM implementation preserves multiple white spaces.

File Inclusion (#include)

A *preprocessor include directive* causes the preprocessor to replace the directive with the contents of the specified file.

A preprocessor **#include** directive has the form:

```
►►──#──include──┬─"──file_name──"───┬──────────────────────────────────►◄
                ├─<──file_name──>───┤
                ├─<──header_name──>─┤
                └─identifiers───────┘
```

In all C and C++ implementations, the preprocessor resolves macros contained in an **#include** directive. After macro replacement, the resulting token sequence must consist of a file name enclosed in either double quotation marks or the characters < and >.

For example:

```
#define MONTH <july.h>
#include MONTH
```

If the file name is enclosed in double quotation marks, for example:

```
#include "payroll.h"
```

the preprocessor treats it as a user-defined file, and searches for the file in a manner defined by the preprocessor.

If the file name is enclosed in angle brackets, for example:

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

it is treated as a system-defined file, and the preprocessor searches for the file in a manner defined by the preprocessor.

The new-line and > characters cannot appear in a file name delimited by < and >. The new-line and " (double quotation marks) character cannot appear in a file name delimited by " and ", although > can.

Declarations that are used by several files can be placed in one file and included with **#include** in each file that uses them. For example, the following file `defs.h` contains several definitions and an inclusion of an additional file of declarations:

```
/* defs.h */
#define TRUE 1
#define FALSE 0
#define BUFFERSIZE 512
#define MAX_ROW 66
#define MAX_COLUMN 80
```

```
int hour;
int min;
int sec;
#include "mydefs.h"
```

You can embed the definitions that appear in defs.h with the following directive:

```
#include "defs.h"
```

In the following example, a **#define** combines several preprocessor macros to define a macro that represents the name of the C standard I/O header file. A **#include** makes the header file available to the program.

```
#define C_IO_HEADER <stdio.h>

/* The following is equivalent to:
 *   #include <stdio.h>
 */

#include C_IO_HEADER
```

Specialized File Inclusion (#include_next)

The preprocessor directive **#include_next** instructs the preprocessor to continue searching for the specified file name, and to include the subsequent instance encountered after the current directory. The syntax of the directive is similar to that of **#include**.

The language feature is an orthogonal extension to C and C++. It extends the techniques available to address the issue of duplicate file names among applications and shared libraries.

ISO Standard Predefined Macro Names

Both C and C++ provide the following predefined macro names as specified in the ISO C language standard. Except for __FILE__ and __LINE__, the value of the predefined macros remains constant throughout the translation unit.

Macro Name **Description**

__DATE__ A character string literal containing the date when the source file was compiled.

The value of __DATE__ changes as the compiler processes any include files that are part of your source program. The date is in the form:

```
"Mmm dd yyyy"
```

where:

Mmm Represents the month in an abbreviated form (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec).

dd Represents the day. If the day is less than 10, the first d is a blank character.

yyyy Represents the year.

__FILE__ A character string literal containing the name of the source file.

The value of __FILE__ changes as the compiler processes include files that are part of your source program. It can be set with the **#line** directive.

__LINE__ An integer representing the current source line number.

The value of `__LINE__` changes during compilation as the compiler processes subsequent lines of your source program. It can be set with the **#line** directive.

`__STDC__`
For C, the integer 1 (one) indicates that the C compiler supports the ISO standard. (When a macro is undefined, it behaves as if it had the integer value 0 when used in a `#if` statement.)

For C++, this macro is predefined to have the value 0 (zero). This indicates that the C++ language is not a proper superset of C, and that the compiler does not conform to ISO C.

`__STDC_HOSTED__`
The value of this C99 macro is 1, indicating that the C compiler is a hosted implementation.

`__STDC_VERSION__`
The integer constant of type **long int**: 199409L for the C89 language level, 199901L for C99.

`__TIME__`
A character string literal containing the time when the source file was compiled.

The value of `__TIME__` changes as the compiler processes any include files that are part of your source program. The time is in the form:

```
"hh:mm:ss"
```

where:

`hh`	Represents the hour.
`mm`	Represents the minutes.
`ss`	Represents the seconds.

`__cplusplus`
For C++ programs, this macro expands to the long integer literal 199711L, indicating that the compiler is a C++ compiler. For C programs, this macro is not defined. Note that this macro name has no trailing underscores.

In addition to the predefined macros required by the language standard, the predefined macro __IBMC__ indicates the level of the C compiler, and the predefined macro __IBMCPP__ indicates that of the C++ compiler.

The value is an integer of the form `VRM`, where

`V`	Represents the version number.
`R`	Represents the release number.
`M`	Represents the modification number.

Related References

- "Line Control (#line)" on page 224
- "Object-Like Macros" on page 211

Conditional Compilation Directives

A *preprocessor conditional compilation directive* causes the preprocessor to conditionally suppress the compilation of portions of source code. These directives test a constant expression or an identifier to determine which tokens the preprocessor should pass on to the compiler and which tokens should be bypassed during preprocessing. The directives are:

- **#if**

- **#ifdef**
- **#else**
- **#ifndef**
- **#elif**
- **#endif**

The preprocessor conditional compilation directive spans several lines:

- The condition specification line (beginning with **#if**, **#ifdef**, or **#ifndef**)
- Lines containing code that the preprocessor passes on to the compiler if the condition evaluates to a nonzero value (optional)
- The **#elif** line (optional)
- Lines containing code that the preprocessor passes on to the compiler if the condition evaluates to a nonzero value (optional)
- The **#else** line (optional)
- Lines containing code that the preprocessor passes on to the compiler if the condition evaluates to zero (optional)
- The preprocessor **#endif** directive

For each **#if**, **#ifdef**, and **#ifndef** directive, there are zero or more **#elif** directives, zero or one **#else** directive, and one matching **#endif** directive. All the matching directives are considered to be at the same nesting level.

You can nest conditional compilation directives. In the following directives, the first **#else** is matched with the **#if** directive.

```
#ifdef MACNAME
                 /*  tokens added if MACNAME is defined */
#   if TEST <=10
                 /* tokens added if MACNAME is defined and TEST <= 10 */
#   else
                 /* tokens added if MACNAME is defined and TEST >  10 */
#   endif
#else
                 /*  tokens added if MACNAME is not defined */
#endif
```

Each directive controls the block immediately following it. A block consists of all the tokens starting on the line following the directive and ending at the next conditional compilation directive at the same nesting level.

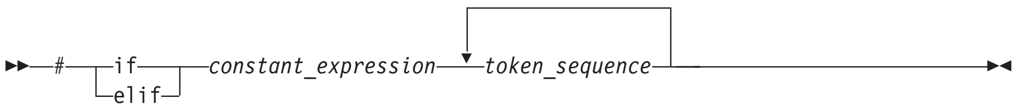
Each directive is processed in the order in which it is encountered. If an expression evaluates to zero, the block following the directive is ignored.

When a block following a preprocessor directive is to be ignored, the tokens are examined only to identify preprocessor directives within that block so that the conditional nesting level can be determined. All tokens other than the name of the directive are ignored.

Only the first block whose expression is nonzero is processed. The remaining blocks at that nesting level are ignored. If none of the blocks at that nesting level has been processed and there is a **#else** directive, the block following the **#else** directive is processed. If none of the blocks at that nesting level has been processed and there is no **#else** directive, the entire nesting level is ignored.

#if, #elif

The **#if** and **#elif** directives compare the value of *constant_expression* to zero:



If the constant expression evaluates to a nonzero value, the lines of code that immediately follow the condition are passed on to the compiler.

If the expression evaluates to zero and the conditional compilation directive contains a preprocessor **#elif** directive, the source text located between the **#elif** and the next **#elif** or preprocessor **#else** directive is selected by the preprocessor to be passed on to the compiler. The **#elif** directive cannot appear after the preprocessor **#else** directive.

All macros are expanded, any `defined()` expressions are processed and all remaining identifiers are replaced with the token `0`.

The *constant_expression* that is tested must be integer constant expressions with the following properties:
- No casts are performed.
- Arithmetic is performed using **long int** values.
- The *constant_expression* can contain defined macros. No other identifiers can appear in the expression.
- The *constant_expression* can contain the unary operator **defined**. This operator can be used only with the preprocessor keyword **#if** or **#elif**. The following expressions evaluate to 1 if the *identifier* is defined in the preprocessor, otherwise to `0`:

  ```
  defined identifier
  defined(identifier)
  ```

  For example:

  ```
  #if defined(TEST1) || defined(TEST2)
  ```

Note: If a macro is not defined, a value of `0` (zero) is assigned to it. In the following example, `TEST` must be a macro identifier:

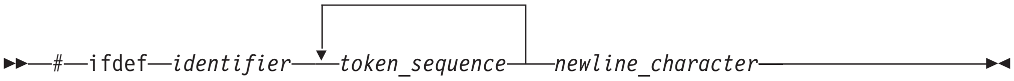
```
#if TEST >= 1
   printf("i = %d\n", i);
   printf("array[i] = %d\n", array[i]);
#elif TEST < 0
   printf("array subscript out of bounds \n");
#endif
```

#ifdef

The **#ifdef** directive checks for the existence of macro definitions.

If the identifier specified is defined as a macro, the lines of code that immediately follow the condition are passed on to the compiler.

The preprocessor **#ifdef** directive has the form:



The following example defines MAX_LEN to be 75 if EXTENDED is defined for the preprocessor. Otherwise, MAX_LEN is defined to be 50.

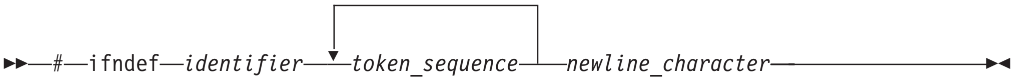
```
#ifdef EXTENDED
#   define MAX_LEN 75
#else
#   define MAX_LEN 50
#endif
```

#ifndef

The **#ifndef** directive checks whether a macro is not defined.

If the identifier specified is not defined as a macro, the lines of code immediately follow the condition are passed on to the compiler.

The preprocessor **#ifndef** directive has the form:



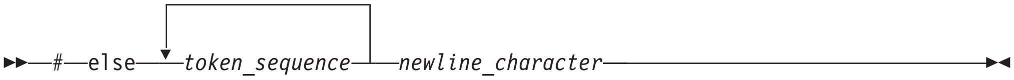
An identifier must follow the **#ifndef** keyword. The following example defines MAX_LEN to be 50 if EXTENDED is not defined for the preprocessor. Otherwise, MAX_LEN is defined to be 75.

```
#ifndef EXTENDED
#   define MAX_LEN 50
#else
#   define MAX_LEN 75
#endif
```

#else

If the condition specified in the **#if**, **#ifdef**, or **#ifndef** directive evaluates to 0, and the conditional compilation directive contains a preprocessor **#else** directive, the lines of code located between the preprocessor **#else** directive and the preprocessor **#endif** directive is selected by the preprocessor to be passed on to the compiler.

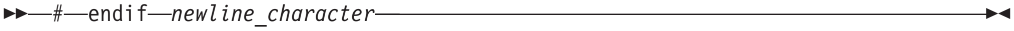
The preprocessor **#else** directive has the form:



#endif

The preprocessor **#endif** directive ends the conditional compilation directive.

It has the form:



Examples of Conditional Compilation Directives

The following example shows how you can nest preprocessor conditional compilation directives:

```
#if defined(TARGET1)
#   define SIZEOF_INT 16
#   ifdef PHASE2
#      define MAX_PHASE 2
#   else
#      define MAX_PHASE 8
#   endif
#elif defined(TARGET2)
#   define SIZEOF_INT 32
#   define MAX_PHASE 16
#else
#   define SIZEOF_INT 32
#   define MAX_PHASE 32
#endif
```

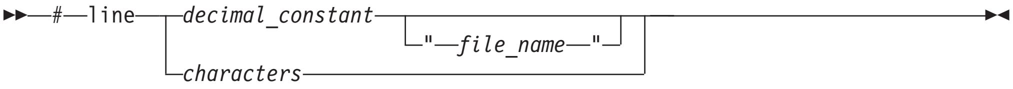
The following program contains preprocessor conditional compilation directives:

```
/**
 ** This example contains preprocessor
 ** conditional compilation directives.
 **/

#include <stdio.h>

int main(void)
{
   static int array[ ] = { 1, 2, 3, 4, 5 };
   int i;

   for (i = 0; i <= 4; i++)
   {
      array[i] *= 2;

#if TEST >= 1
   printf("i = %d\n", i);
   printf("array[i] = %d\n",
   array[i]);
#endif

   }
   return(0);
}
```

Line Control (#line)

A *preprocessor line control directive* supplies line numbers for compiler messages. It causes the compiler to view the line number of the next source line as the specified number.

A preprocessor **#line** directive has the form:



In order for the compiler to produce meaningful references to line numbers in preprocessed source, the preprocessor inserts **#line** directives where necessary (for example, at the beginning and after the end of included text).

A file name specification enclosed in double quotation marks can follow the line number. If you specify a file name, the compiler views the next line as part of the specified file. If you do not specify a file name, the compiler views the next line as part of the current source file.

In all C and C++ implementations, the token sequence on a **#line** directive is subject to macro replacement. After macro replacement, the resulting character sequence must consist of a decimal constant, optionally followed by a file name enclosed in double quotation marks.

Example of the #line Directive

You can use **#line** control directives to make the compiler provide more meaningful error messages. The following program uses **#line** control directives to give each function an easily recognizable line number:

```
/**
 ** This example illustrates #line directives.
 **/

#include <stdio.h>
#define LINE200 200

int main(void)
{
   func_1();
   func_2();
}

#line 100
func_1()
{
   printf("Func_1 - the current line number is %d\n",_ _LINE_ _);
}

#line LINE200
func_2()
{
   printf("Func_2 - the current line number is %d\n",_ _LINE_ _);
}
```

This program produces the following output:

```
Func_1 - the current line number is 102
Func_2 - the current line number is 202
```

Null Directive (#)

The *null directive* performs no action. It consists of a single # on a line of its own.

The null directive should not be confused with the # operator or the character that starts a preprocessor directive.

In the following example, if MINVAL is a defined macro name, no action is performed. If MINVAL is not a defined identifier, it is defined 1.

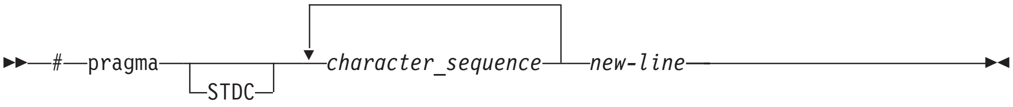
```
#ifdef MINVAL
  #
#else
  #define MINVAL 1
#endif
```

Related References

- "# Operator" on page 215

Pragma Directives (#pragma)

A *pragma* is an implementation-defined instruction to the compiler. It has the general form:



where *character_sequence* is a series of characters giving a specific compiler instruction and arguments, if any. The token STDC indicates a standard pragma; consequently, no macro substitution takes place on the directive. The *new-line* character must terminate a pragma directive.

The *character_sequence* on a pragma is subject to macro substitutions. For example,

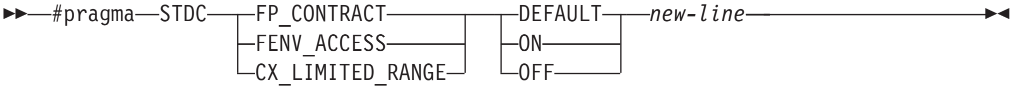
```
#define
XX_ISO_DATA
isolated_call(LG_ISO_DATA)
// ...
#pragma XX_ISO_DATA
```

More than one pragma construct can be specified on a single **#pragma** directive. The compiler ignores unrecognized pragmas.

The available pragmas are discussed in *XL C/C++ Compiler Reference*.

Standard Pragmas

▶ C A *standard pragma* is a pragma preprocessor directive for which the C Standard defines the syntax and semantics and for which no macro replacement is performed. A standard pragma must be one of the following:

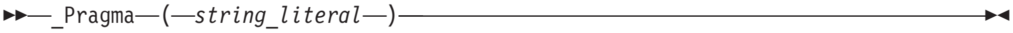


The default for #pragma STDC CX_LIMITED_RANGE is OFF.

The C standard pragmas are discussed in *XL C/C++ Compiler Reference*.

The _Pragma Operator

▶ C The unary operator _Pragma allows a preprocessor macro to be contained in a pragma directive. A _Pragma expression has the following form:



The *string_literal* may be prefixed with L, making it a wide-string literal.

The string literal is destringized and tokenized. The resulting sequence of tokens is processed as if it appeared in a pragma directive. For example:

```
_Pragma ( "align(power)" )
```

would be equivalent to

```
#pragma align(power)
```

Chapter 10. Namespaces

C++ A *namespace* is an optionally named scope. You declare names inside a namespace as you would for a class or an enumeration. You can access names declared inside a namespace the same way you access a nested class name by using the scope resolution (**::**) operator. However namespaces do not have the additional features of classes or enumerations. The primary purpose of the namespace is to add an additional identifier (the name of the namespace) to a name.

Related References
- “C++ Scope Resolution Operator ::” on page 107

Defining Namespaces

C++ In order to uniquely identify a namespace, use the **namespace** keyword.

Syntax – namespace

```
►►—namespace—┬──────────┬—{—namespace_body—}——————————————————————►◄
             └identifier┘
```

The *identifier* in an original namespace definition is the name of the namespace. The identifier may not be previously defined in the declarative region in which the original namespace definition appears, except in the case of extending namespace. If an identifier is not used, the namespace is an *unnamed namespace*.

Related References
- “Unnamed Namespaces” on page 231

Declaring Namespaces

C++ The identifier used for a namespace name should be unique. It should not be used previously as a global identifier.

```
namespace Raymond {
  // namespace body here...
  }
```

In this example, Raymond is the identifier of the namespace. If you intend to access a namespace’s elements, the namespace’s identifier must be known in all translation units.

Related References
- “Global Scope” on page 3

Creating a Namespace Alias

C++ An alternate name can be used in order to refer to a specific namespace identifier.

```
namespace INTERNATIONAL_BUSINESS_MACHINES {
  void f();
}

namespace IBM = INTERNATIONAL_BUSINESS_MACHINES;
```

In this example, the `IBM` identifier is an alias for `INTERNATIONAL_BUSINESS_MACHINES`. This is useful for referring to long namespace identifiers.

If a namespace name or alias is declared as the name of any other entity in the same declarative region, a compiler error will result. Also, if a namespace name defined at global scope is declared as the name of any other entity in any global scope of the program, a compiler error will result.

Related References
- "Global Scope" on page 3

Creating an Alias for a Nested Namespace

▶ C++ Namespace definitions hold declarations. Since a namespace definition is a declaration itself, namespace definitions can be nested.

An alias can also be applied to a nested namespace.

```
namespace INTERNATIONAL_BUSINESS_MACHINES {
  int j;
    namespace NESTED_IBM_PRODUCT {
      void a() { j++; }
      int j;
      void b() { j++; }
  }
}
namespace NIBM = INTERNATIONAL_BUSINESS_MACHINES::NESTED_IBM_PRODUCT
```

In this example, the `NIBM` identifier is an alias for the namespace `NESTED_IBM_PRODUCT`. This namespace is nested within the `INTERNATIONAL_BUSINESS_MACHINES` namespace.

Extending Namespaces

▶ C++ Namespaces are extensible. You can add subsequent declarations to a previously defined namespace. Extensions may appear in files separate from or attached to the original namespace definition. For example:

```
namespace X { // namespace definition
  int a;
  int b;
  }

namespace X { // namespace extension
  int c;
  int d;
  }

namespace Y { // equivalent to namespace X
  int a;
  int b;
  int c;
  int d;
  }
```

In this example, namespace X is defined with a and b and later extended with c and d. namespace X now contains all four members. You may also declare all of the required members within one namespace. This method is represented by namespace Y. This namespace contains a, b, c, and d.

Namespaces and Overloading

C++ You can overload functions across namespaces. For example:

```
// Original X.h:
  f(int);

// Original Y.h:
  f(char);

// Original program.c:
  #include "X.h"
  #include "Y.h"

void z()
{
  f('a'); // calls f(char) from Y.h
}
```

Namespaces can be introduced to the previous example without drastically changing the source code.

```
// New X.h:
namespace X {
  f(int);
  }

// New Y.h:
namespace Y {
  f(char);
  }

// New program.c:
  #include "X.h"
  #include "Y.h"

  using namespace X;
  using namespace Y;

void z()
{
  f('a'); // calls f() from Y.h
}
```

In program.c, function void z() calls function f(), which is a member of namespace Y. If you place the using directives in the header files, the source code for program.c remains unchanged.

Related References
- Chapter 11, "Overloading," on page 237

Unnamed Namespaces

C++ A namespace with no identifier before an opening brace produces an *unnamed namespace*. Each translation unit may contain its own unique unnamed namespace. The following example demonstrates how unnamed namespaces are useful.

```
#include <iostream>

using namespace std;

namespace {
   const int i = 4;
   int variable;
   }

int main()
{
   cout << i << endl;
   variable = 100;
   return 0;
}
```

In the previous example, the unnamed namespace permits access to `i` and `variable` without using a scope resolution operator.

The following example illustrates an improper use of unnamed namespaces.

```
#include <iostream>

using namespace std;

namespace {
   const int i = 4;
   }

int i = 2;

int main()
{
   cout << i << endl; // error
   return 0;
}
```

Inside `main`, `i` causes an error because the compiler cannot distinguish between the global name and the unnamed namespace member with the same name. In order for the previous example to work, the namespace must be uniquely identified with an identifier and `i` must specify the namespace it is using.

You can extend an unnamed namespace within the same translation unit. For example:

```
#include <iostream>

using namespace std;

namespace {
   int variable;
   void funct (int);
   }

namespace {
   void funct (int i) { cout << i << endl; }
   }

int main()
{
   funct(variable);
   return 0;
}
```

both the prototype and definition for `funct` are members of the same unnamed namespace.

Note: Items defined in an unnamed namespace have internal linkage. Rather than using the keyword **static** to define items with internal linkage, define them in an unnamed namespace instead.

Related References
- "Program Linkage" on page 6
- "Internal Linkage" on page 6

Namespace Member Definitions

▶ C++ A namespace can define its own members within itself or externally using explicit qualification. The following is an example of a namespace defining a member internally:

```
namespace A {
  void b() { /* definition */ }
}
```

Within namespace `A` member `void b()` is defined internally.

A namespace can also define its members externally using explicit qualification on the name being defined. The entity being defined must already be declared in the namespace and the definition must appear after the point of declaration in a namespace that encloses the declaration's namespace.

The following is an example of a namespace defining a member externally:

```
namespace A {
  namespace B {
    void f();
  }
  void B::f() { /* defined outside of B */ }
}
```

In this example, function `f()` is declared within namespace `B` and defined (outside `B`) in `A`.

Namespaces and Friends

▶ C++ Every name first declared in a namespace is a member of that namespace. If a `friend` declaration in a non-local class first declares a class or function, the friend class or function is a member of the innermost enclosing namespace.

The following is an example of this structure:

```
// f has not yet been defined
void z(int);
namespace A {
  class X {
    friend void f(X);  // A::f is a friend
    };
  // A::f is not visible here
  X x;
  void f(X) { /* definition */}  // f() is defined and known to be a friend
}

using A::x;
```

```
void z()
{
  A::f(x);    // OK
  A::X::f(x); // error: f is not a member of A::X
}
```

In this example, function `f()` can only be called through namespace `A` using the call `A::f(s);`. Attempting to call function `f()` through `class X` using the `A::X::f(x);` call results in a compiler error. Since the friend declaration first occurs in a non-local class, the friend function is a member of the innermost enclosing namespace and may only be accessed through that namespace.

Related References

- “Friends” on page 278

Using Directive

C++ A *using directive* provides access to all namespace qualifiers and the scope operator. This is accomplished by applying the using keyword to a namespace identifier.

Syntax – Using directive

```
►►──using──namespace──name──;──────────────────────────────────────►◄
```

The *name* must be a previously defined namespace. The using directive may be applied at the global and local scope but not the class scope. Local scope takes precedence over global scope by hiding similar declarations.

If a scope contains a using directive that nominates a second namespace and that second namespace contains another using directive, the using directive from the second namespace will act as if it resides within the first scope.

```
namespace A {
  int i;
}
namespace B {
  int i;
  using namespace A;
}
void f()
{
  using namespace B;
  i = 7; // error
}
```

In this example, attempting to initialize `i` within function `f()` causes a compiler error, because function `f()` cannot know which `i` to call; `i` from namespace `A`, or `i` from namespace `B`.

Related References

- “The using Declaration and Class Members” on page 292

The using Declaration and Namespaces

C++ A *using declaration* provides access to a specific namespace member. This is accomplished by applying the using keyword to a namespace name with its corresponding namespace member.

Syntax – Using declaration

```
►►──using──namespace──::──member──────────────────────────────────────────►◄
```

In this syntax diagram, the qualifier name follows the using declaration and the *member* follows the qualifier name. For the declaration to work, the member must be declared inside the given namespace. For example:

```
namespace A {
  int i;
  int k;
  void f;
  void g;
  }

using A::k
```

In this example, the using declaration is followed by `A`, the name of namespace `A`, which is then followed by the scope operator (`::`), and `k`. This format allows `k` to be accessed outside of namespace `A` through a using declaration. After issuing a using declaration, any extension made to that specific namespace will not be known at the point at which the using declaration occurs.

Overloaded versions of a given function must be included in the namespace prior to that given function's declaration. A using declaration may appear at namespace, block and class scope.

Related References

- "The using Declaration and Class Members" on page 292

Explicit Access

▶ C++ To explicitly qualify a member of a namespace, use the namespace identifier with a :: scope resolution operator.

Syntax – Explicit access qualification

```
►►──namespace_name──::──member────────────────────────────────────────────►◄
```

For example:

```
namespace VENDITTI {
  void j()
};

VENDITTI::j();
```

In this example, the scope resolution operator provides access to the function `j` held within namespace `VENDITTI`. The scope resolution operator `::` is used to access identifiers in both global and local namespaces. Any identifier in an application can be accessed with sufficient qualification. Explicit access cannot be applied to an unnamed namespace.

Related References

- "C++ Scope Resolution Operator ::" on page 107

Chapter 11. Overloading

C++ If you specify more than one definition for a function name or an operator in the same scope, you have *overloaded* that function name or operator.

An *overloaded declaration* is a declaration that had been declared with the same name as a previously declared declaration in the same scope, except that both declarations have different types.

If you call an overloaded function name or operator, the compiler determines the most appropriate definition to use by comparing the argument types you used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called *overload resolution*.

Overloading Functions

C++ You overload a function name `f` by declaring more than one function with the name `f` in the same scope. The declarations of `f` must differ from each other by the types and/or the number of arguments in the argument list. When you call an overloaded function named `f`, the correct function is selected by comparing the argument list of the function call with the parameter list of each of the overloaded candidate functions with the name `f`. A *candidate function* is a function that can be called based on the context of the call of the overloaded function name.

Consider a function `print`, which displays an **int**. As shown in the following example, you can overload the function `print` to display other types, for example, **double** and **char***. You can have three functions with the same name, each performing a similar operation on a different data type:

```
#include <iostream>
using namespace std;

void print(int i) {
  cout << " Here is int " << i << endl;
}
void print(double  f) {
  cout << " Here is float " << f << endl;
}

void print(char* c) {
  cout << " Here is char* " << c << endl;
}

int main() {
  print(10);
  print(10.10);
  print("ten");
}
```

The following is the output of the above example:

```
 Here is int 10
 Here is float 10.1
 Here is char* ten
```

Restrictions on Overloaded Functions

C++ You cannot overload the following function declarations if they appear in the same scope. Note that this list applies only to explicitly declared functions and those that have been introduced through **using** declarations:

- Function declarations that differ only by return type. For example, you cannot declare the following declarations:

  ```
  int f();
  float f();
  ```
- Member function declarations that have the same name and the same parameter types, but one of these declarations is a static member function declaration. For example, you cannot declare the following two member function declarations of `f()`:

  ```
  struct A {
    static int f();
    int f();
  };
  ```
- Member function template declarations that have the same name, the same parameter types, and the same template parameter lists, but one of these declarations is a static template member function declaration.
- Function declarations that have equivalent parameter declarations. These declarations are not allowed because they would be declaring the same function.
- Function declarations with parameters that differ only by the use of **typedef** names that represent the same type. Note that a **typedef** is a synonym for another type, not a separate type. For example, the following two declarations of `f()` are declarations of the same function:

  ```
  typedef int I;
  void f(float, int);
  void f(float, I);
  ```
- Function declarations with parameters that differ only because one is a pointer and the other is an array. For example, the following are declarations of the same function:

  ```
  f(char*);
  f(char[10]);
  ```

  The first array dimension is insignificant when differentiating parameters; all other array dimensions are significant. For example, the following are declarations of the same function:

  ```
  g(char(*)[20]);
  g(char[5][20]);
  ```

  The following two declarations are *not* equivalent:

  ```
  g(char(*)[20]);
  g(char(*)[40]);
  ```
- Function declarations with parameters that differ only because one is a function type and the other is a pointer to a function of the same type. For example, the following are declarations of the same function:

  ```
  void f(int(float));
  void f(int (*)(float));
  ```
- Function declarations with parameters that differ only because of cv-qualifiers **const**, **volatile**, and **restrict**. This restriction only applies if any of these qualifiers appears at the outermost level of a parameter type specification. For example, the following are declarations of the same function:

  ```
  int f(int);
  int f(const int);
  int f(volatile int);
  ```

Note that you can differentiate parameters with **const**, **volatile** and **restrict** qualifiers if you apply them *within* a parameter type specification. For example, the following declarations are *not* equivalent because **const** and **volatile** qualify **int**, rather than *, and thus are not at the outermost level of the parameter type specification.

```
void g(int*);
void g(const int*);
void g(volatile int*);
```

The following declarations are also not equivalent:

```
void g(float&);
void g(const float&);
void g(volatile float&);
```

- Function declarations with parameters that differ only because their default arguments differ. For example, the following are declarations of the same function:

```
void f(int);
void f(int i = 10);
```

- Multiple functions with `extern "C"` language-linkage and the same name, regardless of whether their parameter lists are different.

Overloading Operators

C++ You can redefine or overload the function of most built-in operators in C++. These operators can be overloaded globally or on a class-by-class basis. Overloaded operators are implemented as functions and can be member functions or global functions.

An overloaded operator is called an *operator function*. You declare an operator function with the keyword **operator** preceding the operator. Overloaded operators are distinct from overloaded functions, but like overloaded functions, they are distinguished by the number and types of operands used with the operator.

Consider the standard + (plus) operator. When this operator is used with operands of different standard types, the operators have slightly different meanings. For example, the addition of two integers is not implemented in the same way as the addition of two floating-point numbers. C++ allows you to define your own meanings for the standard C++ operators when they are applied to class types. In the following example, a class called `complx` is defined to model complex numbers, and the + (plus) operator is redefined in this class to add two complex numbers.

```
// This example illustrates overloading the plus (+) operator.

#include <iostream>
using namespace std;

class complx
{
      double real,
             imag;
public:
      complx( double real = 0., double imag = 0.); // constructor
      complx operator+(const complx&) const;       // operator+()
};

// define constructor
complx::complx( double r, double i )
{
      real = r; imag = i;
}
```

```
// define overloaded + (plus) operator
complx complx::operator+ (const complx& c) const
{
      complx result;
      result.real = (this->real + c.real);
      result.imag = (this->imag + c.imag);
      return result;
}

int main()
{
      complx x(4,4);
      complx y(6,6);
      complx z = x + y; // calls complx::operator+()
}
```

You can overload any of the following operators:

+	**−**	*****	**/**	**%**	**^**	**&**	**|**	**~**
!	**=**	**<**	**>**	**+=**	**−=**	***=**	**/=**	**%=**
^=	**&=**	**|=**	**<<**	**>>**	**<<=**	**>>=**	**==**	**!=**
<=	**>=**	**&&**	**||**	**++**	**−−**	**,**	**−>***	**−>**
( )	**[ ]**	**new**	**delete**	**new[]**	**delete[]**			

where () is the function call operator and [] is the subscript operator.

You can overload both the unary and binary forms of the following operators:

+	**-**	*****	**&**

You cannot overload the following operators:

.	**.***	**::**	**?:**

You cannot overload the preprocessor symbols **#** and **##**.

An operator function can be either a nonstatic member function, or a nonmember function with at least one parameter that has class, reference to class, enumeration, or reference to enumeration type.

You cannot change the precedence, grouping, or the number of operands of an operator.

An overloaded operator (except for the function call operator) cannot have default arguments or an ellipsis in the argument list.

You must declare the overloaded **=**, **[]**, **()**, and **->** operators as nonstatic member functions to ensure that they receive lvalues as their first operands.

The operators **new**, **delete**, **new[]**, and **delete[]** do not follow the general rules described in this section.

All operators except the **=** operator are inherited.

Related References
- “Free Store” on page 323

Overloading Unary Operators

C++ You overload a unary operator with either a nonstatic member function that has no parameters, or a nonmember function that has one parameter. Suppose a unary operator @ is called with the statement `@t`, where `t` is an object of type `T`. A nonstatic member function that overloads this operator would have the following form:

```
 return_type operator@()
```

A nonmember function that overloads the same operator would have the following form:

```
return_type operator@(T)
```

An overloaded unary operator may return any type.

The following example overloads the **!** operator:

```
#include <iostream>
using namespace std;

struct X { };

void operator!(X) {
  cout << "void operator!(X)" << endl;
}

struct Y {
  void operator!() {
    cout << "void Y::operator!()" << endl;
  }
};

struct Z { };

int main() {
  X ox; Y oy; Z oz;
  !ox;
  !oy;
//  !oz;
}
```

The following is the output of the above example:

```
void operator!(X)
void Y::operator!()
```

The operator function call `!ox` is interpreted as `operator!(X)`. The call `!oy` is interpreted as `Y::operator!()`. (The compiler would not allow `!oz` because the **!** operator has not been defined for class `Z`.)

Related References
- "Unary Expressions" on page 119

Overloading Increment and Decrement

C++ You overload the prefix increment operator ++ with either a nonmember function operator that has one argument of class type or a reference to class type, or with a member function operator that has no arguments.

In the following example, the increment operator is overloaded in both ways:

```
class X {
public:

  // member prefix ++x
  void operator++() { }
};

class Y { };

// non-member prefix ++y
void operator++(Y&) { }

int main() {
  X x;
  Y y;

  // calls x.operator++()
  ++x;

  // explicit call, like ++x
  x.operator++();

  // calls operator++(y)
  ++y;

  // explicit call, like ++y
  operator++(y);
}
```

The postfix increment operator ++ can be overloaded for a class type by declaring a nonmember function operator `operator++()` with two arguments, the first having class type and the second having type **int**. Alternatively, you can declare a member function operator `operator++()` with one argument having type **int**. The compiler uses the **int** argument to distinguish between the prefix and postfix increment operators. For implicit calls, the default value is zero.

For example:

```
class X {
public:

  // member postfix x++
  void operator++(int) { };
};

class Y { };

// nonmember postfix y++
void operator++(Y&, int) { };

int main() {
  X x;
  Y y;

  // calls x.operator++(0)
  // default argument of zero is supplied by compiler
  x++;
  // explicit call to member postfix x++
  x.operator++(0);

  // calls operator++(y, 0)
  y++;
  // explicit call to non-member postfix y++
  operator++(y, 0);
}
```

The prefix and postfix decrement operators follow the same rules as their increment counterparts.

Related References
- “Increment ++” on page 120
- “Decrement ––” on page 120

Overloading Binary Operators

C++ You overload a binary unary operator with either a nonstatic member function that has one parameter, or a nonmember function that has two parameters. Suppose a binary operator @ is called with the statement `t @ u`, where `t` is an object of type `T`, and `u` is an object of type `U`. A nonstatic member function that overloads this operator would have the following form:

```
 return_type operator@(T)
```

A nonmember function that overloads the same operator would have the following form:

```
return_type operator@(T, U)
```

An overloaded binary operator may return any type.

The following example overloads the * operator:

```
struct X {

  // member binary operator
  void operator*(int) { }
};

// non-member binary operator
void operator*(X, float) { }

int main() {
  X x;
  int y = 10;
  float z = 10;

  x * y;
  x * z;
}
```

The call `x * y` is interpreted as `x.operator*(y)`. The call `x * z` is interpreted as `operator*(x, z)`.

Related References
- “Binary Expressions” on page 133

Overloading Assignments

C++ You overload the assignment operator, **operator=**, with a nonstatic member function that has only one parameter. You cannot declare an overloaded assignment operator that is a nonmember function. The following example shows how you can overload the assignment operator for a particular class:

```
struct X {
  int data;
  X& operator=(X& a) { return a; }
  X& operator=(int a) {
    data = a;
    return *this;
  }
};

int main() {
  X x1, x2;
  x1 = x2;      // call x1.operator=(x2)
  x1 = 5;       // call x1.operator=(5)
}
```

The assignment `x1 = x2` calls the copy assignment operator `X& X::operator=(X&)`. The assignment `x1 = 5` calls the copy assignment operator `X& X::operator=(int)`. The compiler implicitly declares a copy assignment operator for a class if you do not define one yourself. Consequently, the copy assignment operator (**operator=**) of a derived class hides the copy assignment operator of its base class.

However, you can declare any copy assignment operator as virtual. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct A {
  A& operator=(char) {
    cout << "A& A::operator=(char)" << endl;
    return *this;
  }
  virtual A& operator=(const A&) {
    cout << "A& A::operator=(const A&)" << endl;
    return *this;
  }
};

struct B : A {
    B& operator=(char) {
      cout << "B& B::operator=(char)" << endl;
      return *this;
    }
    virtual B& operator=(const A&) {
      cout << "B& B::operator=(const A&)" << endl;
      return *this;
    }
};

struct C : B { };

int main() {
  B b1;
  B b2;
  A* ap1 = &b1;
  A* ap2 = &b1;
  *ap1 = 'z';
  *ap2 = b2;

  C c1;
//  c1 = 'z';
}
```

The following is the output of the above example:

```
A& A::operator=(char)
B& B::operator=(const A&)
```

The assignment `*ap1 = 'z'` calls `A& A::operator=(char)`. Because this operator has not been declared **virtual**, the compiler chooses the function based on the type of the pointer `ap1`. The assignment `*ap2 = b2` calls `B& B::operator=(const &A)`. Because this operator has been declared **virtual**, the compiler chooses the function based on the type of the object that the pointer `ap1` points to. The compiler would not allow the assignment `c1 = 'z'` because the implicitly declared copy assignment operator declared in class `C` hides `B& B::operator=(char)`.

Related References

- “Copy Assignment Operators” on page 333
- “Assignment Expressions” on page 144

Overloading Function Calls

C++ The function call operator, when overloaded, does not modify how functions are called. Rather, it modifies how the operator is to be interpreted when applied to objects of a given type.

You overload the function call operator, **operator()**, with a nonstatic member function that has any number of parameters. If you overload a function call operator for a class its declaration will have the following form:

```
return_type operator()(parameter_list)
```

Unlike all other overloaded operators, you can provide default arguments and ellipses in the argument list for the function call operator.

The following example demonstrates how the compiler interprets function call operators:

```
struct A {
  void operator()(int a, char b, ...) { }
  void operator()(char c, int d = 20) { }
};

int main() {
  A a;
  a(5, 'z', 'a', 0);
  a('z');
//  a();
}
```

The function call `a(5, 'z', 'a', 0)` is interpreted as `a.operator()(5, 'z', 'a', 0)`. This calls `void A::operator()(int a, char b, ...)`. The function call `a('z')` is interpreted as `a.operator()('z')`. This calls `void A::operator()(char c, int d = 20)`. The compiler would not allow the function call `a()` because its argument list does not match any function call parameter list defined in class `A`.

The following example demonstrates an overloaded function call operator:

```
class Point {
private:
  int x, y;
public:
  Point() : x(0), y(0) { }
  Point& operator()(int dx, int dy) {
    x += dx;
    y += dy;
    return *this;
  }
};

int main() {
  Point pt;

  // Offset this coordinate x with 3 points
  // and coordinate y with 2 points.
  pt(3, 2);
}
```

The above example reinterprets the function call operator for objects of class `Point`. If you treat an object of `Point` like a function and pass it two integer arguments, the function call operator will add the values of the arguments you passed to `Point::x` and `Point::y` respectively.

Related References

- "Function Call Operator ( )" on page 109

Overloading Subscripting

C++ You overload **operator[]** with a nonstatic member function that has only one parameter. The following example is a simple array class that has an overloaded subscripting operator. The overloaded subscripting operator throws an exception if you try to access the array outside of its specified bounds:

```
#include <iostream>
using namespace std;

template <class T> class MyArray {
private:
  T* storage;
  int size;
public:
  MyArray(int arg = 10) {
    storage = new T[arg];
    size = arg;
  }

  ~MyArray() {
    delete[] storage;
    storage = 0;
  }

  T& operator[](const int location) throw (const char *);
};

template <class T> T& MyArray<T>::operator[](const int location)
  throw (const char *) {
    if (location < 0 || location >= size) throw "Invalid array access";
    else return storage[location];
}

int main() {
  try {
     MyArray<int> x(13);
     x[0] = 45;
     x[1] = 2435;
     cout << x[0] << endl;
     cout << x[1] << endl;
     x[13] = 84;
  }
  catch (const char* e) {
    cout << e << endl;
  }
}
```

The following is the output of the above example:

```
45
2435
Invalid array access
```

The expression `x[1]` is interpreted as `x.operator[](1)` and calls `int& MyArray<int>::operator[](const int)`.

Related References

- "Array Subscripting Operator [ ]" on page 111

Overloading Class Member Access

C++ You overload **operator->** with a nonstatic member function that has no parameters. The following example demonstrates how the compiler interprets overloaded class member access operators:

```
struct Y {
  void f() { };
};

struct X {
 Y* ptr;
 Y* operator->() {
   return ptr;
 };
};

int main() {
  X x;
  x->f();
}
```

The statement `x->f()` is interpreted as `(x.operator->())->f()`.

The **operator->** is used (often in conjunction with the pointer-dereference operator) to implement ″smart pointers.″ These pointers are objects that behave like normal pointers except they perform other tasks when you access an object through them, such as automatic object deletion (either when the pointer is destroyed, or the pointer is used to point to another object), or reference counting (counting the number of smart pointers that point to the same object, then automatically deleting the object when that count reaches zero).

One example of a smart pointer is included in the C++ Standard Library called `auto_ptr`. You can find it in the `<memory>` header. The `auto_ptr` class implements automatic object deletion.

Related References

- "Arrow Operator −>" on page 112

Overload Resolution

C++ The process of selecting the most appropriate overloaded function or operator is called *overload resolution*.

Suppose that f is an overloaded function name. When you call the overloaded function f(), the compiler creates a set of *candidate functions*. This set of functions includes all of the functions named f that can be accessed from the point where you called f(). The compiler may include as a candidate function an alternative representation of one of those accessible functions named f to facilitate overload resolution.

After creating a set of candidate functions, the compiler creates a set of *viable functions*. This set of functions is a subset of the candidate functions. The number of parameters of each viable function agrees with the number of arguments you used to call f().

The compiler chooses the *best viable function*, the function declaration that the C++ run time will use when you call f(), from the set of viable functions. The compiler does this by *implicit conversion sequences*. An implicit conversion sequence is the sequence of conversions required to convert an argument in a function call to the type of the corresponding parameter in a function declaration. The implicit conversion sequences are ranked; some implicit conversion sequences are better than others. The compiler tries to find one viable function in which all of its parameters have either better or equal-ranked implicit conversion sequences than all of the other viable functions. The viable function that the compiler finds is the best viable function. The compiler will not allow a program in which the compiler was able to find more than one best viable function.

When a variable length array is a function parameter, the leftmost array dimension does not distinguish functions among candidate functions. In the following, the second definition of f is not allowed because void f(int []) has already been defined.

```
void f(int a[*]) {}
void f(int a[5]) {} // illegal
```

However, array dimensions other than the leftmost in a variable length array do differentiate candidate functions when the variable length array is a function parameter. For example, the overload set for function f might comprise the following:

```
void f(int a[][5]) {}
void f(int a[][4]) {}
void f(int a[][g]) {}   // assume g is a global int
```

but cannot include

```
void f(int a[][g2]) {} // illegal, assuming g2 is a global int
```

because having candidate functions with second-level array dimensions g and g2 creates ambiguity about which function f should be called: neither g nor g2 is known at compile time.

You can override an exact match by using an explicit cast. In the following example, the second call to f() matches with f(void*):

```
void f(int) { };
void f(void*) { };

int main() {
   f(0xaabb);            // matches f(int);
   f((void*) 0xaabb);    // matches f(void*)
}
```

Implicit Conversion Sequences

C++ An *implicit conversion sequence* is the sequence of conversions required to convert an argument in a function call to the type of the corresponding parameter in a function declaration.

The compiler will try to determine an implicit conversion sequence for each argument. It will then categorize each implicit conversion sequence in one of three categories and rank them depending on the category. The compiler will not allow any program in which it cannot find an implicit conversion sequence for an argument.

The following are the three categories of conversion sequences in order from best to worst:
- Standard conversion sequences
- User-defined conversion sequences
- Ellipsis conversion sequences

Note: Two standard conversion sequences or two user-defined conversion sequences may have different ranks.

Standard Conversion Sequences

Standard conversion sequences are categorized in one of three ranks. The ranks are listed in order from best to worst:
- Exact match: This rank includes the following conversions:
  - Identity conversions
  - Lvalue-to-rvalue conversions
  - Array-to-pointer conversions
  - Qualification conversions
- Promotion: This rank includes integral and floating point promotions.
- Conversion: This rank includes the following conversions:
  - Integral and floating-point conversions
  - Floating-integral conversions
  - Pointer conversions
  - Pointer-to-member conversions
  - Boolean conversions

The compiler ranks a standard conversion sequence by its worst-ranked standard conversion. For example, if a standard conversion sequence has a floating-point conversion, then that sequence has conversion rank.

User-Defined Conversion Sequences

A *user-defined conversion sequence* consists of the following:
- A standard conversion sequence
- A user-defined conversion
- A second standard conversion sequence

A user-defined conversion sequence `A` is better than a user-defined conversion sequence `B` if the both have the same user-defined conversion function or constructor, and the second standard conversion sequence of `A` is better than the second standard conversion sequence of `B`.

Ellipsis Conversion Sequences

An *ellipsis conversion sequence* occurs when the compiler matches an argument in a function call with a corresponding ellipsis parameter.

Related References
- "Lvalue-to-Rvalue Conversions" on page 150
- "Pointer Conversions" on page 152
- "Qualification Conversions" on page 154
- "Integral Conversions" on page 151
- "Floating-Point Conversions" on page 152
- "Boolean Conversions" on page 151

Resolving Addresses of Overloaded Functions

▶ C++ If you use an overloaded function name f without any arguments, that name can refer to a function, a pointer to a function, a pointer to member function, or a specialization of a function template. Because you did not provide any arguments, the compiler cannot perform overload resolution the same way it would for a function call or for the use of an operator. Instead, the compiler will try to choose the best viable function that matches the type of one of the following expressions, depending on where you have used f:
- An object or reference you are initializing
- The left side of an assignment
- A parameter of a function or a user-defined operator
- The return value of a function, operator, or conversion
- An explicit type conversion

If the compiler chose a declaration of a nonmember function or a static member function when you used f, the compiler matched the declaration with an expression of type pointer-to-function or reference-to-function. If the compiler chose a declaration of a nonstatic member function, the compiler matched that declaration with an expression of type pointer-to-member function. The following example demonstrates this:

```
struct X {
  int f(int) { return 0; }
  static int f(char) { return 0; }
};

int main() {
  int (X::*a)(int) = &X::f;
//  int (*b)(int) = &X::f;
}
```

The compiler will not allow the initialization of the function pointer b. No nonmember function or static function of type **int(int)** has been declared.

If f is a template function, the compiler will perform template argument deduction to determine which template function to use. If successful, it will add that function to the list of viable functions. If there is more than one function in this set, including a non-template function, the compiler will eliminate all template functions from the set and choose the non-template function. If there are only template functions in this set, the compiler will choose the most specialized template function. The following example demonstrates this:

```
template<class T> int f(T) { return 0; }
template<> int f(int) { return 0; }
int f(int) { return 0; }
int main() {
  int (*a)(int) = f;
  a(1);
}
```

The function call `a(1)` calls `int f(int)`.

Related References

- “Pointers to Functions” on page 185
- “Pointers to Members” on page 268
- “Function Templates” on page 347
- “Explicit Specialization” on page 358

Chapter 12. Classes

C++ A *class* is a mechanism for creating user-defined data types. It is similar to the C language structure data type. In C, a structure is composed of a set of data members. In C++, a class type is like a C structure, except that a class is composed of a set of data members and a set of operations that can be performed on the class.

In C++, a class type can be declared with the keywords **union**, **struct**, or **class**. A union object can hold any one of a set of named members. Structure and class objects hold a complete set of members. Each class type represents a unique set of class members including data members, member functions, and other type names. The default access for members depends on the class key:

- The members of a class declared with the keyword **class** are private by default. A class is inherited privately by default.
- The members of a class declared with the keyword **struct** are public by default. A structure is inherited publicly by default.
- The members of a union (declared with the keyword **union**) are public by default. A union cannot be used as a base class in derivation.

Once you create a class type, you can declare one or more objects of that class type. For example:

```
class X
{
     /* define class members here */
};
int main()
{
      X xobject1;       // create an object of class type X
      X xobject2;       // create another object of class type X
}
```

You may have *polymorphic* classes in C++. Polymorphism is the ability to use a function name that appears in different classes (related by inheritance), without knowing exactly the class the function belongs to at compile time.

C++ allows you to redefine standard operators and functions through the concept of overloading. Operator overloading facilitates data abstraction by allowing you to use classes as easily as built-in types.

Declaring Class Types

C++ A class declaration creates a unique type class name.

A *class specifier* is a type specifier used to declare a class. Once a class specifier has been seen and its members declared, a class is considered to be defined even if the member functions of that class are not yet defined. A class specifier has the following form:

Syntax –

```
►►──┬─class──┬──class_name──┬───────────────┬──{──┬─────────────┬──}──────────►◄
    ├─struct─┤              └─:──base_clause─┘     └─member_list─┘
    └─union──┘
```

The *class_name* is a unique identifier that becomes a reserved word within its scope. Once a class name is declared, it hides other declarations of the same name within the enclosing scope.

The *member_list* specifies the class members, both data and functions, of the class *class_name*. If the *member_list* of a class is empty, objects of that class have a nonzero size. You can use a *class_name* within the *member_list* of the class specifier itself as long as the size of the class is not required.

The *base_clause* specifies the base class or classes from which the class *class_name* inherits members. If the *base_clause* is not empty, the class *class_name* is called a *derived class*.

A *structure* is a class declared with the *class_key* **struct**. The members and base classes of a structure are public by default. A *union* is a class declared with the *class_key* **union**. The members of a union are public by default; a union holds only one data member at a time.

An *aggregate class* is a class that has no user-defined constructors, no private or protected non-static data members, no base classes, and no virtual functions.

Using Class Objects

▶ C++ You can use a class type to create instances or *objects* of that class type. For example, you can declare a class, structure, and union with class names X, Y, and Z respectively:

```
class X {
  // members of class X
};

struct Y {
  // members of struct Y
};

union Z {
  // members of union Z
};
```

You can then declare objects of each of these class types. Remember that classes, structures, and unions are all types of C++ classes.

```
int main()
{
      X xobj;      // declare a class object of class type X
      Y yobj;      // declare a struct object of class type Y
      Z zobj;      // declare a union object of class type Z
}
```

In C++, unlike C, you do not need to precede declarations of class objects with the keywords **union**, **struct**, and **class** unless the name of the class is hidden. For example:

```
struct Y { /* ... */ };
class X { /* ... */ };
int main ()
{
      int X;                   // hides the class name X
      Y yobj;                  // valid
      X xobj;                  // error, class name X is hidden
      class X xobj;            // valid
}
```

When you declare more than one class object in a declaration, the declarators are treated as if declared individually. For example, if you declare two objects of class S in a single declaration:

```
class S { /* ... */ };
int main()
{
      S S,T; // declare two objects of class type S
}
```

this declaration is equivalent to:

```
class S { /* ... */ };
int main()
{
      S S;
      class S T;       // keyword class is required
                       // since variable S hides class type S
}
```

but is not equivalent to:

```
class S { /* ... */ };
int main()
{
      S S;
      S T;             // error, S class type is hidden
}
```

You can also declare references to classes, pointers to classes, and arrays of classes. For example:

```
class X { /* ... */ };
struct Y { /* ... */ };
union Z { /* ... */ };
int main()
{
      X xobj;
      X &xref = xobj;           // reference to class object of type X
      Y *yptr;                  // pointer to struct object of type Y
      Z zarray[10];             // array of 10 union objects of type Z
}
```

Objects of class types that are not copy restricted can be assigned, passed as arguments to functions, and returned by functions.

Classes and Structures

▶ C++ The C++ class is an extension of the C language structure. Because the only difference between a structure and a class is that structure members have public access by default and class members have private access by default, you can use the keywords **class** or **struct** to define equivalent classes.

For example, in the following code fragment, the class X is equivalent to the structure Y:

```
class X {

  // private by default
  int a;

public:

  // public member function
  int f() { return a = 5; };
};

struct Y {

  // public by default
  int f() { return a = 5; };

private:

  // private data member
  int a;
};
```

If you define a structure and then declare an object of that structure using the keyword **class**, the members of the object are still public by default. In the following example, main() has access to the members of obj_X even though obj_X has been declared using an elaborated type specifier that uses the class key **class**:

```
#include <iostream>
using namespace std;

struct X {
 int a;
 int b;
};

class X obj_X;

int main() {
  obj_X.a = 0;
  obj_X.b = 1;
  cout << "Here are a and b: " << obj_X.a << " " << obj_X.b << endl;
}
```

The following is the output of the above example:

```
Here are a and b: 0 1
```

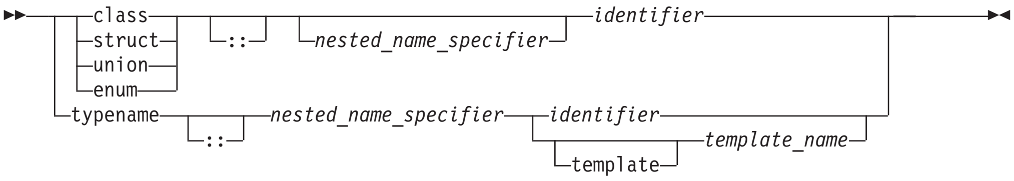
Related References
- "Structures" on page 54

Scope of Class Names

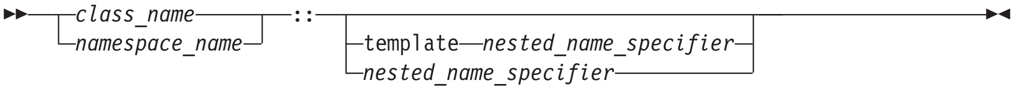
C++ A class declaration introduces the class name into the scope where it is declared. Any class, object, function or other declaration of that name in an enclosing scope is hidden.

If a class name is declared in the same scope as a function, enumerator, or object with the same name, you must refer to that class using an *elaborated type specifier*:

Syntax – Elaborated Type Specifier



Syntax – Nested Name Specifier



The following example must use an elaborated type specifier to refer to class A because this class is hidden by the definition of the function A():

```
class A { };

void A (class A*) { };

int main()
{
      class A* x;
      A(x);
}
```

The declaration class A* x is an elaborated type specifier. Declaring a class with the same name of another function, enumerator, or object as demonstrated above is not recommended.

An elaborated type specifier can also be used in the incomplete declaration of a class type to reserve the name for a class type within the current scope.

Incomplete Class Declarations

C++ An *incomplete class declaration* is a class declaration that does not define any class members. You cannot declare any objects of the class type or refer to the members of a class until the declaration is complete. However, an incomplete declaration allows you to make specific references to a class prior to its definition as long as the size of the class is not required.

For example, you can define a pointer to the structure first in the definition of the structure second. Structure first is declared in an incomplete class declaration prior to the definition of second, and the definition of oneptr in structure second does not require the size of first:

```
struct first;           // incomplete declaration of struct first

struct second           // complete declaration of struct second
{
      first* oneptr;    // pointer to struct first refers to
                        // struct first prior to its complete
                        // declaration

      first one;        // error, you cannot declare an object of
                        // an incompletely declared class type
      int x, y;
};
```

```
struct first            // complete declaration of struct first
{
      second two;       // define an object of class type second
      int z;
};
```

However, if you declare a class with an empty member list, it is a complete class declaration. For example:

```
class X;                // incomplete class declaration
class Z {};             // empty member list
class Y
{
public:
      X yobj;           // error, cannot create an object of an
                        // incomplete class type
      Z zobj;           // valid
};
```

Related References

- "Class Member Lists" on page 263

Nested Classes

C++ A *nested class* is declared within the scope of another class. The name of a nested class is local to its enclosing class. Unless you use explicit pointers, references, or object names, declarations in a nested class can only use visible constructs, including type names, static members, and enumerators from the enclosing class and global variables.

Member functions of a nested class follow regular access rules and have no special access privileges to members of their enclosing classes. Member functions of the enclosing class have no special access to members of a nested class. The following example demonstrates this:

```
class A {
  int x;

  class B { };

  class C {

    // The compiler cannot allow the following
    // declaration because A::B is private:
    //   B b;

    int y;
    void f(A* p, int i) {

    // The compiler cannot allow the following
    // statement because A::x is private:
    //   p->x = i;

    }
  };

  void g(C* p) {

    // The compiler cannot allow the following
    // statement because C::y is private:
    //   int z = p->y;
  }
};

int main() { }
```

The compiler would not allow the declaration of object b because class A::B is private. The compiler would not allow the statement p->x = i because A::x is private. The compiler would not allow the statement int z = p->y because C::y is private.

You can define member functions and static data members of a nested class in namespace scope. For example, in the following code fragment, you can access the static members x and y and member functions f() and g() of the nested class nested by using a qualified type name. Qualified type names allow you to define a **typedef** to represent a qualified class name. You can then use the **typedef** with the :: (scope resolution) operator to refer to a nested class or class member, as shown in the following example:

```
class outside
{
public:
      class nested
      {
      public:
            static int x;
            static int y;
            int f();
            int g();
      };
};
int outside::nested::x = 5;
int outside::nested::f() { return 0; };

typedef outside::nested outnest;       // define a typedef
int outnest::y = 10;                   // use typedef with ::
int outnest::g() { return 0; };
```

However, using a typedef to represent a nested class name hides information and may make the code harder to understand.

You cannot use a typedef name in an elaborated type specifier. To illustrate, you cannot use the following declaration in the above example:

```
  class outnest obj;
```

A nested class may inherit from private members of its enclosing class. The following example demonstrates this:

```
class A {
private:
  class B { };
  B *z;

  class C : private B {
  private:
      B y;
//      A::B y2;
      C *x;
//      A::C *x2;
    };
};
```

The nested class A::C inherits from A::B. The compiler does not allow the declarations A::B y2 and A::C *x2 because both A::B and A::C are private.

Local Classes

C++ A *local class* is declared within a function definition. Declarations in a local class can only use type names, enumerations, static variables from the enclosing scope, as well as external variables and functions.

For example:

```
int x;                         // global variable
void f()                       // function definition
{
      static int y;            // static variable y can be used by
                               // local class
      int x;                   // auto variable x cannot be used by
                               // local class
      extern int g();          // extern function g can be used by
                               // local class

      class local              // local class
      {
            int g() { return x; }      // error, local variable x
                                       // cannot be used by g
            int h() { return y; }      // valid,static variable y
            int k() { return ::x; }    // valid, global x
            int l() { return g(); }    // valid, extern function g
      };
}

int main()
{
      local* z;                // error: the class local is not visible
      // ...}
```

Member functions of a local class have to be defined within their class definition, if they are defined at all. As a result, member functions of a local class are inline functions. Like all member functions, those defined within the scope of a local class do not need the keyword **inline**.

A local class cannot have static data members. In the following example, an attempt to define a static member of a local class causes an error:

```
void f()
{
    class local
    {
       int f();              // error, local class has noninline
                             // member function
       int g() {return 0;}   // valid, inline member function
       static int a;         // error, static is not allowed for
                             // local class
       int b;                // valid, nonstatic variable
    };
}
//      . . .
```

An enclosing function has no special access to members of the local class.

Related References
- “Member Functions” on page 264
- “Inline Functions” on page 186

Local Type Names

C++ Local type names follow the same scope rules as other names. Type names defined within a class declaration have class scope and cannot be used outside their class without qualification.

If you use a class name, **typedef** name, or a constant name that is used in a type name, in a class declaration, you cannot redefine that name after it is used in the class declaration.

For example:

```
int main ()
{
      typedef double db;
      struct st
      {
            db x;
            typedef int db; // error
            db y;
      };
}
```

The following declarations are valid:

```
typedef float T;
class s {
      typedef int T;
      void f(const T);
};
```

Here, function `f()` takes an argument of type `s::T`. However, the following declarations, where the order of the members of `s` has been reversed, cause an error:

```
typedef float T;
class s {
      void f(const T);
      typedef int T;
};
```

In a class declaration, you cannot redefine a name that is not a class name, or a **typedef** name to a class name or **typedef** name once you have used that name in the class declaration.

Related References

- “Scope” on page 1
- “typedef” on page 42

Scope of Class Names

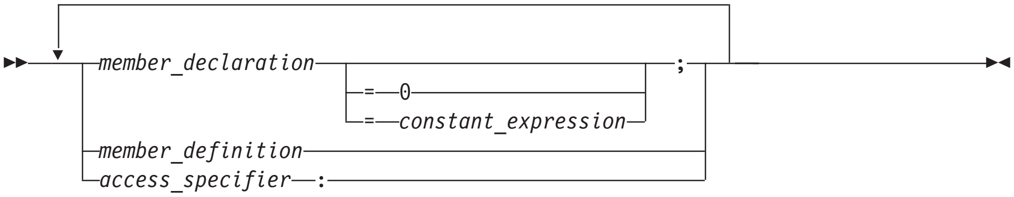
Chapter 13. Class Members and Friends

C++ This section discusses the declaration of class members with respect to the information hiding mechanism and how a class can grant functions and classes access to its nonpublic members by the use of the friend mechanism. C++ expands the concept of information hiding to include the notion of having a public class interface but a private implementation. It is the mechanism for limiting direct access to the internal representation of a class type by functions in a program.

Class Member Lists

C++ An optional *member list* declares subobjects called *class members*. Class members can be data, functions, nested types, and enumerators.

Syntax – Class Member List



The member list follows the class name and is placed between braces. The following applies to member lists, and members of member lists:

- A *member_declaration* or a *member_definition* may be a declaration or definition of a data member, member function, nested type, or enumeration. (The enumerators of a enumeration defined in a class member list are also members of the class.)
- A member list is the only place where you can declare class members.
- Friend declarations are not class members but must appear in member lists.
- The member list in a class definition declares all the members of a class; you cannot add members elsewhere.
- You cannot declare a member twice in a member list.
- You may declare a data member or member function as **static** but not **auto**, **extern**, or **register**.
- You may declare a nested class, a member class template, or a member function, and define it outside the class.
- You must define static data members outside the class.
- Nonstatic members that are class objects must be objects of previously defined classes; a class A cannot contain an object of class A, but it can contain a pointer or reference to an object of class A.
- You must specify all dimensions of a nonstatic array member.

A *constant initializer* (= *constant_expression*) may only appear in a class member of integral or enumeration type that has been declared **static**.

A *pure specifier* (= 0) indicates that a function has no definition. It is only used with member functions declared as **virtual** and replaces the function definition of a member function in the member list.

An *access specifier* is one of **public**, **private**, or **protected**.

A *member declaration* declares a class member for the class containing the declaration.

The order of allocation of nonstatic class members separated by an *access_specifier* is implementation-dependent. The compiler allocates class members in the order in which they are declared.

Suppose `A` is a name of a class. The following class members of `A` must have a name different from `A`:
- All data members
- All type members
- All enumerators of enumerated type members
- All members of all anonymous union members

Data Members

▶ C++ Data members include members that are declared with any of the fundamental types, as well as other types, including pointer, reference, array types, bit fields, and user-defined types. You can declare a data member the same way as a variable, except that explicit initializers are not allowed inside the class definition. However, a const static data member of integral or enumeration type may have an explicit initializer.

If an array is declared as a nonstatic class member, you must specify all of the dimensions of the array.

A class can have members that are of a class type or are pointers or references to a class type. Members that are of a class type must be of a class type that has been previously declared. An incomplete class type can be used in a member declaration as long as the size of the class is not needed. For example, a member can be declared that is a pointer to an incomplete class type.

A class `X` cannot have a member that is of type `X`, but it can contain pointers to `X`, references to `X`, and static objects of `X`. Member functions of `X` can take arguments of type `X` and have a return type of `X`. For example:

```
class X
{
      X();
      X *xptr;
      X &xref;
      static X xcount;
      X xfunc(X);
};
```

Member Functions

▶ C++ *Member functions* are operators and functions that are declared as members of a class. Member functions do not include operators and functions declared with the **friend** specifier. These are called *friends* of a class. You can declare a member function as **static**; this is called a *static member function*. A member function that is not declared as **static** is called a *nonstatic member function*.

Suppose that you create an object named x of class A, and class A has a nonstatic member function f(). If you call the function x.f(), the keyword **this** in the body of f() is the address of x.

The definition of a member function is within the scope of its enclosing class. The body of a member function is analyzed after the class declaration so that members of that class can be used in the member function body, even if the member function definition appears before the declaration of that member in the class member list. When the function add() is called in the following example, the data variables a, b, and c can be used in the body of add().

```
class x
{
public:
      int add()             // inline member function add
      {return a+b+c;};
private:
      int a,b,c;
};
```

Inline Member Functions

You may either define a member function inside its class definition, or you may define it outside if you have already declared (but not defined) the member function in the class definition.

A member function that is defined inside its class member list is called an *inline member function*. Member functions containing a few lines of code are usually declared inline. In the above example, add() is an inline member function. If you define a member function outside of its class definition, it must appear in a namespace scope enclosing the class definition. You must also qualify the member function name using the scope resolution (::) operator.

An equivalent way to declare an inline member function is to either declare it in the class with the **inline** keyword (and define the function outside of its class) or to define it outside of the class declaration using the **inline** keyword.

In the following example, member function Y::f() is an inline member function:

```
struct Y {
private:
  char a*;
public:
  char* f() { return a; }
};
```

The following example is equivalent to the previous example; Y::f() is an inline member function:

```
struct Y {
private:
  char a*;
public:
  char* f();
};

inline char* Z::f() { return a; }
```

The **inline** specifier does not affect the linkage of a member or nonmember function: linkage is external by default.

Member Functions of Local Classes

Member functions of a local class must be defined within their class definition. As a result, member functions of a local class are implicitly inline functions. These inline member functions have no linkage.

Related References
- “Friends” on page 278
- “Static Member Functions” on page 275
- Chapter 7, “Functions,” on page 159
- “Inline Functions” on page 186
- “Local Classes” on page 260

const and volatile Member Functions

C++ A member function declared with the **const** qualifier can be called for constant and nonconstant objects. A nonconstant member function can only be called for a nonconstant object. Similarly, a member function declared with the **volatile** qualifier can be called for volatile and nonvolatile objects. A nonvolatile member function can only be called for a nonvolatile object.

Related References
- “Type Qualifiers” on page 71
- “The const Type Qualifier” on page 73

Virtual Member Functions

C++ Virtual member functions are declared with the keyword **virtual**. They allow dynamic binding of member functions. Because all virtual functions must be member functions, virtual member functions are simply called *virtual functions*.

If the definition of a virtual function is replaced by a pure specifier in the declaration of the function, the function is said to be declared pure. A class that has at least one pure virtual function is called an *abstract class*.

Related References
- “Virtual Functions” on page 302
- “Abstract Classes” on page 308

Special Member Functions

C++ *Special member functions* are used to create, destroy, initialize, convert, and copy class objects. These include the following:
- Constructors
- Destructors
- Conversion constructors
- Conversion functions
- Copy constructors

Related References
- Chapter 15, “Special Member Functions,” on page 311

Member Scope

C++ Member functions and static members can be defined outside their class declaration if they have already been declared, but not defined, in the class member list. Nonstatic data members are defined when an object of their class is created. The declaration of a static data member is not a definition. The declaration of a member function is a definition if the body of the function is also given.

Whenever the definition of a class member appears outside of the class declaration, the member name must be qualified by the class name using the `::` (scope resolution) operator.

The following example defines a member function outside of its class declaration.

```
#include <iostream>
using namespace std;

struct X {
  int a, b ;

  // member function declaration only
  int add();
};

// global variable
int a  = 10;

// define member function outside its class declaration
int X::add() { return a + b; }

int main() {
  int answer;
  X xobject;
  xobject.a = 1;
  xobject.b = 2;
  answer = xobject.add();
  cout << xobject.a << " + " << xobject.b << " = " << answer << endl;
}
```

The output for this example is: `1 + 2 = 3`

All member functions are in class scope even if they are defined outside their class declaration. In the above example, the member function `add()` returns the data member `a`, not the global variable `a`.

The name of a class member is local to its class. Unless you use one of the class access operators, `.` (dot), or `->` (arrow), or `::` (scope resolution) operator, you can only use a class member in a member function of its class and in nested classes. You can only use types, enumerations and static members in a nested class without qualification with the `::` operator.

The order of search for a name in a member function body is:
1. Within the member function body itself
2. Within all the enclosing classes, including inherited members of those classes
3. Within the lexical scope of the body declaration

The search of the enclosing classes, including inherited members, is demonstrated in the following example:

```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class Z : A {
      class Y : B {
            class X : C { int f(); /* ... */ };
      };
};
int Z::Y::X f()
{
      char j;
      return 0;
}
```

In this example, the search for the name j in the definition of the function f follows this order:

1. In the body of the function f
2. In X and in its base class C
3. In Y and in its base class B
4. In Z and in its base class A
5. In the lexical scope of the body of f. In this case, this is global scope.

Note that when the containing classes are being searched, only the definitions of the containing classes and their base classes are searched. The scope containing the base class definitions (global scope, in this example) is not searched.

Related References

- "Class Scope" on page 4

Pointers to Members

▶ C++ Pointers to members allow you to refer to nonstatic members of class objects. You cannot use a pointer to member to point to a static class member because the address of a static member is not associated with any particular object. To point to a static class member, you must use a normal pointer.

You can use pointers to member functions in the same manner as pointers to functions. You can compare pointers to member functions, assign values to them, and use them to call member functions. Note that a member function does not have the same type as a nonmember function that has the same number and type of arguments and the same return type.

Pointers to members can be declared and used as shown in the following example:

```
#include <iostream>
using namespace std;

class X {
public:
  int a;
  void f(int b) {
    cout << "The value of b is "<< b << endl;
  }
};

int main() {

  // declare pointer to data member
  int X::*ptiptr = &X::a;

  // declare a pointer to member function
  void (X::* ptfptr) (int) = &X::f;

  // create an object of class type X
  X xobject;

  // initialize data member
  xobject.*ptiptr = 10;

  cout << "The value of a is " << xobject.*ptiptr << endl;

  // call member function
  (xobject.*ptfptr) (20);
}
```

The output for this example is:

```
The value of a is 10
The value of b is 20
```

To reduce complex syntax, you can declare a **typedef** to be a pointer to a member. A pointer to a member can be declared and used as shown in the following code fragment:

```
typedef int X::*my_pointer_to_member;
typedef void (X::*my_pointer_to_function) (int);

int main() {
  my_pointer_to_member ptiptr = &X::a;
  my_pointer_to_function ptfptr = &X::f;
  X xobject;
  xobject.*ptiptr = 10;
  cout << "The value of a is " << xobject.*ptiptr << endl;
  (xobject.*ptfptr) (20);
}
```

The pointer to member operators `.*` and `->*` are used to bind a pointer to a member of a specific class object. Because the precedence of `()` (function call operator) is higher than `.*` and `->*`, you must use parentheses to call the function pointed to by `ptf`.

Related References
- “C++ Pointer to Member Operators .* −>*” on page 142
- “Objects” on page 35

The this Pointer

C++ The keyword **this** identifies a special type of pointer. Suppose that you create an object named `x` of class `A`, and class `A` has a nonstatic member function `f()`. If you call the function `x.f()`, the keyword **this** in the body of `f()` is the address of `x`. You cannot declare the **this** pointer or make assignments to it.

A static member function does not have a **this** pointer.

The type of the **this** pointer for a member function of a class type `X`, is `X* const`. If the member function is declared with the **const** qualifier, the type of the **this** pointer for that member function for class `X`, is `const X* const`. If the member function is declared with the **volatile** qualifier, the type of the **this** pointer for that member function for class `X` is `volatile X* const`. For example, the compiler will not allow the following:

```
struct A {
  int a;
  int f() const { return a++; }
};
```

The compiler will not allow the statement `a++` in the body of function `f()`. In the function `f()`, the **this** pointer is of type `A* const`. The function `f()` is trying to modify part of the object to which **this** points.

The **this** pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions.

For example, you can refer to the particular class object that a member function is called for by using the **this** pointer in the body of the member function. The following code example produces the output `a = 5`:

```
#include <iostream>
using namespace std;

struct X {
private:
  int a;
public:
  void Set_a(int a) {

    // The 'this' pointer is used to retrieve 'xobj.a'
    // hidden by the automatic variable 'a'
    this->a = a;
  }
   void Print_a() { cout << "a = " << a << endl; }
};

int main() {
  X xobj;
  int a = 5;
  xobj.Set_a(a);
  xobj.Print_a();
}
```

In the member function `Set_a()`, the statement `this->a = a` uses the **this** pointer to retrieve `xobj.a` hidden by the automatic variable `a`.

Unless a class member name is hidden, using the class member name is equivalent to using the class member name with the **this** pointer and the class member access operator (**->**).

The example in the first column of the following table shows code that uses class members without the **this** pointer. The code in the second column uses the variable `THIS` to simulate the first column's hidden use of the **this** pointer:

Code without using this pointer	Equivalent code, the `THIS` variable simulating the hidden use of the this pointer
`#include <string>`	`#include <string>`
`#include <iostream>`	`#include <iostream>`
`using namespace std;`	`using namespace std;`
`struct X {`	`struct X {`
`private:`	`private:`
`  int len;`	`  int len;`
`  char *ptr;`	`  char *ptr;`
`public:`	`public:`
`  int GetLen() {`	`  int GetLen (X* const THIS) {`
`    return len;`	`    return THIS->len;`
`  }`	`  }`
`  char * GetPtr() {`	`  char * GetPtr (X* const THIS) {`
`    return ptr;`	`    return THIS->ptr;`
`  }`	`  }`
`  X& Set(char *);`	`  X& Set(X* const, char *);`
`  X& Cat(char *);`	`  X& Cat(X* const, char *);`
`  X& Copy(X&);`	`  X& Copy(X* const, X&);`
`  void Print();`	`  void Print(X* const);`
`};`	`};`
`X& X::Set(char *pc) {`	`X& X::Set(X* const THIS, char *pc) {`
`  len = strlen(pc);`	`  THIS->len = strlen(pc);`
`  ptr = new char[len];`	`  THIS->ptr =  new char[THIS->len];`
`  strcpy(ptr, pc);`	`  strcpy(THIS->ptr, pc);`
`  return *this;`	`  return *THIS;`
`}`	`}`
`X& X::Cat(char *pc) {`	`X& X::Cat(X* const THIS, char *pc) {`
`  len += strlen(pc);`	`  THIS->len += strlen(pc);`
`  strcat(ptr,pc);`	`  strcat(THIS->ptr, pc);`
`  return *this;`	`  return *THIS;`
`}`	`}`
`X& X::Copy(X& x) {`	`X& X::Copy(X* const THIS, X& x) {`
`  Set(x.GetPtr());`	`  THIS->Set(THIS, x.GetPtr(&x));`
`  return *this;`	`  return *THIS;`
`}`	`}`
`void X::Print() {`	`void X::Print(X* const THIS) {`
`  cout << ptr << endl;`	`  cout << THIS->ptr << endl;`
`}`	`}`
`int main() {`	`int main() {`
`  X xobj1;`	`  X xobj1;`
`  xobj1.Set("abcd")`	`  xobj1.Set(&xobj1 , "abcd")`
`       .Cat("efgh");`	`       .Cat(&xobj1 , "efgh");`
`  xobj1.Print();`	`  xobj1.Print(&xobj1);`
`  X xobj2;`	`  X xobj2;`
`  xobj2.Copy(xobj1)`	`  xobj2.Copy(&xobj2 , xobj1)`
`       .Cat("ijkl");`	`       .Cat(&xobj2 , "ijkl");`
`  xobj2.Print();`	`  xobj2.Print(&xobj2);`
`}`	`}`

Both examples produces the following output:

```
abcdefgh
abcdefghijkl
```

Related References

- "Overloading Assignments" on page 243

- “Copy Constructors” on page 332

Static Members

C++ Class members can be declared using the storage class specifier **static** in the class member list. Only one copy of the static member is shared by all objects of a class in a program. When you declare an object of a class having a static member, the static member is not part of the class object.

A typical use of static members is for recording data common to all objects of a class. For example, you can use a static data member as a counter to store the number of objects of a particular class type that are created. Each time a new object is created, this static data member can be incremented to keep track of the total number of objects.

You access a static member by qualifying the class name using the `::` (scope resolution) operator. In the following example, you can refer to the static member `f()` of class type `X` as `X::f()` even if no object of type `X` is ever declared:

```
struct X {
  static int f();
};

int main() {
  X::f();
}
```

Related References
- “static Storage Class Specifier” on page 40
- “Class Member Lists” on page 263

Using the Class Access Operators with Static Members

C++ You do not have to use the class member access syntax to refer to a static member; to access a static member `s` of class `X`, you could use the expression `X::s`. The following example demonstrates accessing a static member:

```
#include <iostream>
using namespace std;

struct A {
  static void f() { cout << "In static function A::f()" << endl; }
};

int main() {

  // no object required for static member
  A::f();

  A a;
  A* ap = &a;
  a.f();
  ap->f();
}
```

The three statements `A::f()`, `a.f()`, and `ap->f()` all call the same static member function `A::f()`.

You can directly refer to a static member in the same scope of its class, or in the scope of a class derived from the static member’s class. The following example demonstrates the latter case (directly referring to a static member in the scope of a class derived from the static member's class):

```
#include <iostream>
using namespace std;

int g() {
   cout << "In function g()" << endl;
   return 0;
}

class X {
   public:
      static int g() {
         cout << "In static member function X::g()" << endl;
         return 1;
      }
};

class Y: public X {
   public:
      static int i;
};

int Y::i = g();

int main() { }
```

The following is the output of the above code:

```
In static member function X::g()
```

The initialization `int Y::i = g()` calls `X::g()`, not the function `g()` declared in the global namespace.

A static member can be referred to independently of any association with a class object because there is only one static member shared by all objects of a class. A static member will exist even if no objects of its class have been declared.

Related References
- "Dot Operator ." on page 112
- "Arrow Operator −>" on page 112

Static Data Members

C++ Only one copy of a static data member of a class exists; it is shared with all objects of that class.

Static data members of a class in namespace scope have external linkage. Static data members follow the usual class access rules, except that they can be initialized in file scope. Static data members and their initializers can access other static private and protected members of their class. The initializer for a static data member is in the scope of the class declaring the member.

A static data member can be of any type except for **void** or **void** qualified with **const** or **volatile**.

The declaration of a static data member in the member list of a class is not a definition. The definition of a static data member is equivalent to an external variable definition. You must define the static member outside of the class declaration in namespace scope.

For example:

```
class X
{
public:
      static int i;
};
int X::i = 0; // definition outside class declaration
```

Once you define a static data member, it exists even though no objects of the static data member's class exist. In the above example, no objects of class X exist even though the static data member X::i has been defined.

The following example shows how you can initialize static members using other static members, even though these members are private:

```
class C {
      static int i;
      static int j;
      static int k;
      static int l;
      static int m;
      static int n;
      static int p;
      static int q;
      static int r;
      static int s;
      static int f() { return 0; }
      int a;
public:
      C() { a = 0; }
      };

C c;
int C::i = C::f();    // initialize with static member function
int C::j = C::i;      // initialize with another static data member
int C::k = c.f();     // initialize with member function from an object
int C::l = c.j;       // initialize with data member from an object
int C::s = c.a;       // initialize with nonstatic data member
int C::r = 1;         // initialize with a constant value

class Y : private C {} y;

int C::m = Y::f();
int C::n = Y::r;
int C::p = y.r;       // error
int C::q = y.f();     // error
```

The initializations of C::p and C::q cause errors because y is an object of a class that is derived privately from C, and its members are not accessible to members of C.

If a static data member is of **const** integral or **const** enumeration type, you may specify a *constant initializer* in the static data member's declaration. This constant initializer must be an integral constant expression. Note that the constant initializer is not a definition. You still need to define the static member in an enclosing namespace. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct X {
  static const int a = 76;
};

const int X::a;
```

```
int main() {
  cout << X::a << endl;
}
```

The tokens `= 76` at the end of the declaration of static data member `a` is a constant initializer.

You can only have one definition of a static member in a program. Unnamed classes and classes contained within unnamed classes cannot have static data members.

You cannot declare a static data member as **mutable**.

Local classes cannot have static data members.

Static Member Functions

▶ C++ You cannot have static and nonstatic member functions with the same names and the same number and type of arguments.

Like static data members, you may access a static member function `f()` of a class `A` without using an object of class `A`.

A static member function does not have a **this** pointer. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct X {
private:
  int i;
  static int si;
public:
  void set_i(int arg) { i = arg; }
  static void set_si(int arg) { si = arg; }

  void print_i() {
    cout << "Value of i = " << i << endl;
    cout << "Again, value of i = " << this->i << endl;
  }

  static void print_si() {
    cout << "Value of si = " << si << endl;
//    cout << "Again, value of si = " << this->si << endl;
  }

};

int X::si = 77;       // Initialize static data member

int main() {
  X xobj;
  xobj.set_i(11);
  xobj.print_i();

  // static data members and functions belong to the class and
  // can be accessed without using an object of class X
  X::print_si();
  X::set_si(22);
  X::print_si();
}
```

The following is the output of the above example:

```
Value of i = 11
Again, value of i = 11
Value of si = 77
Value of si = 22
```

The compiler does not allow the member access operation `this->si` in function `A::print_si()` because this member function has been declared as static, and therefore does not have a **this** pointer.

You can call a static member function using the **this** pointer of a nonstatic member function. In the following example, the nonstatic member function `printall()` calls the static member function `f()` using the **this** pointer:

```
#include <iostream>
using namespace std;

class C {
  static void f() {
    cout << "Here is i: " << i << endl;
  }
  static int i;
  int j;
public:
  C(int firstj): j(firstj) { }
  void printall();
};

void C::printall() {
  cout << "Here is j: " << this->j << endl;
  this->f();
}

int C::i = 3;

int main() {
  C obj_C(0);
  obj_C.printall();
}
```

The following is the output of the above example:

```
Here is j: 0
Here is i: 3
```

A static member function cannot be declared with the keywords **virtual**, **const**, **volatile**, or **const volatile**.

A static member function can access only the names of static members, enumerators, and nested types of the class in which it is declared. Suppose a static member function `f()` is a member of class `X`. The static member function `f()` cannot access the nonstatic members `X` or the nonstatic members of a base class of `X`.

Related References
- "The this Pointer" on page 269

Member Access

C++ *Member access* determines if a class member is accessible in an expression or declaration. Suppose `x` is a member of class `A`. Class member `x` can be declared to have one of the following levels of accessibility:

- **public**: x can be used anywhere without the access restrictions defined by private or protected.
- **private**: x can be used only by the members and friends of class A.
- **protected**: x can be used only by the members and friends of class A, and the members and friends of classes derived from class A.

Members of classes declared with the keyword **class** are private by default. Members of classes declared with the keyword **struct** or **union** are public by default.

To control the access of a class member, you use one of the *access specifiers* **public**, **private**, or **protected** as a label in a class member list. The following example demonstrates these access specifiers:

```
struct A {
  friend class C;
private:
  int a;
public:
  int b;
protected:
  int c;
};

struct B : A {
  void f() {
    // a = 1;
    b = 2;
    c = 3;
  }
};

struct C {
  void f(A x) {
    x.a = 4;
    x.b = 5;
    x.c = 6;
  }
};

int main() {
  A y;
//  y.a = 7;
  y.b = 8;
//  y.c = 9;

  B z;
//  z.a = 10;
  z.b = 11;
//  z.c = 12;
}
```

The following table lists the access of data members A::a A::b, and A::c in various scopes of the above example:

Scope	**A::a**	**A::b**	**A::c**
function B::f()	No access. Member A::a is private.	Access. Member A::b is public.	Access. Class B inherits from A.
function C::f()	Access. Class C is a friend of A.	Access. Member A::b is public.	Access. Class C is a friend of A.
object y in main()	No access. Member y.a is private.	Access. Member y.a is public.	No access. Member y.c is protected.

Scope	A::a	A::b	A::c
object z in main()	No access. Member z.a is private.	Access. Member z.a is public.	No access. Member z.c is protected.

An access specifier specifies the accessibility of members that follow it until the next access specifier or until the end of the class definition. You can use any number of access specifiers in any order. If you later define a class member within its class definition, its access specification must be the same as its declaration. The following example demonstrates this:

```
class A {
    class B;
  public:
    class B { };
};
```

The compiler will not allow the definition of class B because this class has already been declared as private.

A class member has the same access control regardless whether it has been defined within its class or outside its class.

Access control applies to names. In particular, if you add access control to a typedef name, it affects only the typedef name. The following example demonstrates this:

```
class A {
    class B { };
  public:
    typedef B C;
};

int main() {
  A::C x;
//  A::B y;
}
```

The compiler will allow the declaration A::C x because the typedef name A::C is public. The compiler would not allow the declaration A::B y because A::B is private.

Note that accessibility and visibility are independent. Visibility is based on the scoping rules of C++. A class member can be visible and inaccessible at the same time.

Friends

▶ C++ A friend of a class X is a function or class that is not a member of X, but is granted the same access to X as the members of X. Functions declared with the **friend** specifier in a class member list are called *friend functions* of that class. Classes declared with the **friend** specifier in the member list of another class are called *friend classes* of that class.

A class Y must be defined before any member of Y can be declared a friend of another class.

In the following example, the friend function print is a member of class Y and accesses the private data members a and b of class X.

```
#include <iostream>
using namespace std;

class X;

class Y {
public:
  void print(X& x);
};

class X {
  int a, b;
  friend void Y::print(X& x);
public:
  X() : a(1), b(2) { }
};

void Y::print(X& x) {
  cout << "a is " << x.a << endl;
  cout << "b is " << x.b << endl;
}

int main() {
  X xobj;
  Y yobj;
  yobj.print(xobj);
}
```

The following is the output of the above example:

```
a is 1
b is 2
```

You can declare an entire class as a friend. Suppose class F is a friend of class A. This means that every member function and static data member definition of class F has access to class A.

In the following example, the friend class F has a member function print that accesses the private data members a and b of class X and performs the same task as the friend function print in the above example. Any other members declared in class F also have access to all members of class X:

```
#include <iostream>
using namespace std;

class X {
  int a, b;
  friend class F;
public:
  X() : a(1), b(2) { }
};

class F {
public:
  void print(X& x) {
    cout << "a is " << x.a << endl;
    cout << "b is " << x.b << endl;
  }
};

int main() {
  X xobj;
  F fobj;
  fobj.print(xobj);
}
```

The following is the output of the above example:

```
a is 1
b is 2
```

You must use an elaborated type specifier when you declare a class as a friend. The following example demonstrates this:

```
class F;
class G;
class X {
  friend class F;
  friend G;
};
```

The compiler will warn you that the friend declaration of `G` must be an elaborated class name.

You cannot define a class in a friend declaration. For example, the compiler will not allow the following:

```
class F;
class X {
  friend class F { };
};
```

However, you can define a function in a friend declaration. The class must be a non-local class, function, the function name must be unqualified, and the function has namespace scope. The following example demonstrates this:

```
class A {
  void g();
};

void z() {
  class B {
//    friend void f() { };
  };
}

class C {
//  friend void A::g() { }
  friend void h() { }
};
```

The compiler would not allow the function definition of `f()` or `g()`. The compiler will allow the definition of `h()`.

You cannot declare a friend with a storage class specifier.

Related References
- "Member Access" on page 276
- "Inherited Member Access" on page 290

Friend Scope

C++ The name of a friend function or class first introduced in a friend declaration is not in the scope of the class granting friendship (also called the *enclosing class*) and is not a member of the class granting friendship.

The name of a function first introduced in a friend declaration is in the scope of the first nonclass scope that contains the enclosing class. The body of a function provided in a friend declaration is handled in the same way as a member function defined within a class. Processing of the definition does not start until the end of the outermost enclosing class. In addition, unqualified names in the body of the function definition are searched for starting from the class containing the function definition.

A class that is first declared in a friend declaration is equivalent to an **extern** declaration. For example:

```
class B {};
class A
{
      friend class B; // global class B is a friend of A
};
```

If the name of a friend class has been introduced before the friend declaration, the compiler searches for a class name that matches the name of the friend class beginning at the scope of the friend declaration. If the declaration of a nested class is followed by the declaration of a friend class with the same name, the nested class is a friend of the enclosing class.

The scope of a friend class name is the first nonclass enclosing scope. For example:

```
class A {
   class B { // arbitrary nested class definitions
      friend class C;
   };
};
```

is equivalent to:

```
class C;
class A {
   class B { // arbitrary nested class definitions
      friend class C;
   };
};
```

If the friend function is a member of another class, you need to use the scope resolution operator (::). For example:

```
class A {
public:
  int f() { }
};

class B {
  friend int A::f();
};
```

Friends of a base class are not inherited by any classes derived from that base class. The following example demonstrates this:

```
class A {
  friend class B;
  int a;
};

class B { };

class C : public B {
  void f(A* p) {
//    p->a = 2;
  }
};
```

The compiler would not allow the statement p->a = 2 because class C is not a friend of class A, although C inherits from a friend of A.

Friendship is not transitive. The following example demonstrates this:

```
class A {
  friend class B;
  int a;
};

class B {
  friend class C;
};

class C {
  void f(A* p) {
//    p->a = 2;
  }
};
```

The compiler would not allow the statement p->a = 2 because class C is not a friend of class A, although C is a friend of a friend of A.

If you declare a friend in a local class, and the friend's name is unqualified, the compiler will look for the name only within the innermost enclosing nonclass scope. You must declare a function before declaring it as a friend of a local scope. You do not have to do so with classes. However, a declaration of a friend class will hide a class in an enclosing scope with the same name. The following example demonstrates this:

```
class X { };
void a();

void f() {
  class Y { };
  void b();
  class A {
    friend class X;
    friend class Y;
    friend class Z;
//    friend void a();
    friend void b();
//    friend void c();
  };
  ::X moocow;
//  X moocow2;
}
```

In the above example, the compiler will allow the following statements:
- friend class X: This statement does not declare ::X as a friend of A, but the local class X as a friend, even though this class is not otherwise declared.
- friend class Y: Local class Y has been declared in the scope of f().
- friend class Z: This statement declares the local class Z as a friend of A even though Z is not otherwise declared.
- friend void b(): Function b() has been declared in the scope of f().
- ::X moocow: This declaration creates an object of the nonlocal class ::X.

The compiler would not allow the following statements:
- friend void a(): This statement does not consider function a() declared in namespace scope. Since function a() has not been declared in the scope of f(), the compiler would not allow this statement.

- `friend void c()`: Since function `c()` has not been declared in the scope of `f()`, the compiler would not allow this statement.
- `X moocow2`: This declaration tries to create an object of the local class `X`, not the nonlocal class `::X`. Since local class `X` has not been defined, the compiler would not allow this statement.

Related References
- “Local Classes” on page 260

Friend Access

C++ A friend of a class can access the private and protected members of that class. Normally, you can only access the private members of a class through member functions of that class, and you can only access the protected members of a class through member functions of a class or classes derived from that class.

Friend declarations are not affected by access specifiers.

Related References
- “Member Access” on page 276

Chapter 14. Inheritance

C++ *Inheritance* is a mechanism of reusing and extending existing classes without modifying them.

Inheritance is almost like embedding an object into a class. Suppose that you declare an object x of class A in the class definition of B. As a result, class B will have access to all the public data members and member functions of class A. However, in class B, you have to access the data members and member functions of class A through object x. The following example demonstrates this:

```
#include <iostream>
using namespace std;

class A {
   int data;
public:
   void f(int arg) { data = arg; }
   int g() { return data; }
};

class B {
public:
   A x;
};

int main() {
   B obj;
   obj.x.f(20);
   cout << obj.x.g() << endl;
//   cout << obj.g() << endl;
}
```

In the main function, object obj accesses function A::f() through its data member B::x with the statement obj.x.f(20). Object obj accesses A::g() in a similar manner with the statement obj.x.g(). The compiler would not allow the statement obj.g() because g() is a member function of class A, not class B.

The inheritance mechanism lets you use a statement like obj.g() in the above example. In order for that statement to be legal, g() must be a member function of class B.

Inheritance lets you include the names and definitions of another class's members as part of a new class. The class whose members you want to include in your new class is called a *base class*. Your new class is *derived* from the base class. You new class will contain a *subobject* of the type of the base class. The following example is the same as the previous example except it uses the inheritance mechanism to give class B access to the members of class A:

```
#include <iostream>
using namespace std;

class A {
   int data;
public:
   void f(int arg) { data = arg; }
   int g() { return data; }
};
```

```
class B : public A { };

int main() {
   B obj;
   obj.f(20);
   cout << obj.g() << endl;
}
```

Class A is a base class of class B. The names and definitions of the members of class A are included in the definition of class B; class B inherits the members of class A. Class B is derived from class A. Class B contains a subobject of type A.

You can also add new data members and member functions to the derived class. You can modify the implementation of existing member functions or data by overriding base class member functions or data in the newly derived class.

You may derive classes from other derived classes, thereby creating another level of inheritance. The following example demonstrates this:

```
struct A { };
struct B : A { };
struct C : B { };
```

Class B is a derived class of A, but is also a base class of C. The number of levels of inheritance is only limited by resources.

Multiple inheritance allows you to create a derived class that inherits properties from more than one base class. Because a derived class inherits members from all its base classes, ambiguities can result. For example, if two base classes have a member with the same name, the derived class cannot implicitly differentiate between the two members. Note that, when you are using multiple inheritance, the access to names of base classes may be ambiguous.

A *direct base class* is a base class that appears directly as a base specifier in the declaration of its derived class.

An *indirect base class* is a base class that does not appear directly in the declaration of the derived class but is available to the derived class through one of its base classes. For a given class, all base classes that are not direct base classes are indirect base classes. The following example demonstrates direct and indirect base classes:

```
class A {
  public:
    int x;
};
class B : public A {
  public:
    int y;
};
class C : public B { };
```

Class B is a direct base class of C. Class A is a direct base class of B. Class A is an indirect base class of C. (Class C has x and y as its data members.)

Polymorphic functions are functions that can be applied to objects of more than one type. In C++, polymorphic functions are implemented in two ways:

- Overloaded functions are statically bound at compile time.

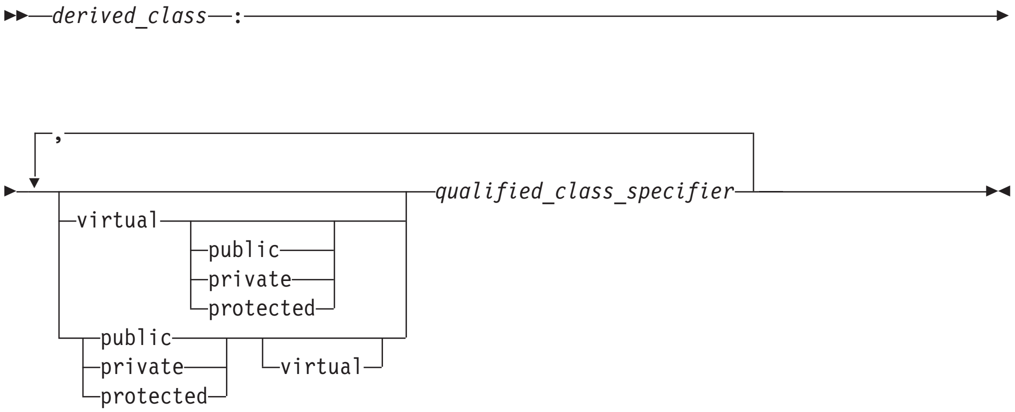
- C++ provides virtual functions. A *virtual function* is a function that can be called for a number of different user-defined types that are related through derivation. Virtual functions are bound dynamically at run time.

Related References
- "Multiple Access" on page 298
- "Multiple Inheritance" on page 296
- "Virtual Functions" on page 302

Derivation

C++ Inheritance is implemented in C++ through the mechanism of derivation. Derivation allows you to derive a class, called a *derived class*, from another class, called a *base class*.

Syntax – Derived Class Derivation



In the declaration of a derived class, you list the base classes of the derived class. The derived class inherits its members from these base classes.

The *qualified_class_specifier* must be a class that has been previously declared in a class declaration.

An *access specifier* is one of **public**, **private**, or **protected**.

The **virtual** keyword can be used to declare virtual base classes.

The following example shows the declaration of the derived class D and the base classes V, B1, and B2. The class B1 is both a base class and a derived class because it is derived from class V and is a base class for D:

```
class V { /* ... */ };
class B1 : virtual public V { /* ... */ };
class B2 { /* ... */ };
class D : public B1, private B2 { /* ... */ };
```

Classes that are declared but not defined are not allowed in base lists.

For example:

```
class X;

// error
class Y: public X { };
```

The compiler will not allow the declaration of class Y because X has not been defined.

When you derive a class, the derived class inherits class members of the base class. You can refer to inherited members (base class members) as if they were members of the derived class. For example:

```
class Base {
public:
  int a,b;
};

class Derived : public Base {
public:
  int c;
};

int main() {
  Derived d;
  d.a = 1;    // Base::a
  d.b = 2;    // Base::b
  d.c = 3;    // Derived::c
}
```

The derived class can also add new class members and redefine existing base class members. In the above example, the two inherited members, a and b, of the derived class d, in addition to the derived class member c, are assigned values. If you redefine base class members in the derived class, you can still refer to the base class members by using the :: (scope resolution) operator. For example:

```
#include <iostream>
using namespace std;

class Base {
public:
  char* name;
  void display() {
    cout << name << endl;
  }
};

class Derived: public Base {
public:
  char* name;
  void display() {
    cout << name << ", " << Base::name << endl;
   }
};

int main() {
  Derived d;
  d.name = "Derived Class";
  d.Base::name = "Base Class";

  // call Derived::display()
  d.display();

  // call Base::display()
  d.Base::display();
}
```

The following is the output of the above example:

```
Derived Class, Base Class
Base Class
```

You can manipulate a derived class object as if it were a base class object. You can use a pointer or a reference to a derived class object in place of a pointer or reference to its base class. For example, you can pass a pointer or reference to a derived class object D to a function expecting a pointer or reference to the base class of D. You do not need to use an explicit cast to achieve this; a standard conversion is performed. You can implicitly convert a pointer to a derived class to point to an accessible unambiguous base class. You can also implicitly convert a reference to a derived class to a reference to a base class.

The following example demonstrates a standard conversion from a pointer to a derived class to a pointer to a base class:

```
#include <iostream>
using namespace std;

class Base {
public:
  char* name;
  void display() {
    cout << name << endl;
  }
};

class Derived: public Base {
public:
  char* name;
  void display() {
    cout << name << ", " << Base::name << endl;
   }
};

int main() {
  Derived d;
  d.name = "Derived Class";
  d.Base::name = "Base Class";

  Derived* dptr = &d;

  // standard conversion from Derived* to Base*
  Base* bptr = dptr;

  // call Base::display()
  bptr->display();
}
```

The following is the output of the above example:

```
Base Class
```

The statement Base* bptr = dptr converts a pointer of type Derived to a pointer of type Base.

The reverse case is not allowed. You cannot implicitly convert a pointer or a reference to a base class object to a pointer or reference to a derived class. For example, the compiler will not allow the following code if the classes Base and Class are defined as in the above example:

```
int main() {
  Base b;
  b.name = "Base class";

  Derived* dptr = &b;
}
```

The compiler will not allow the statement `Derived* dptr = &b` because the statement is trying to implicitly convert a pointer of type `Base` to a pointer of type `Derived`.

If a member of a derived class and a member of a base class have the same name, the base class member is hidden in the derived class. If a member of a derived class has the same name as a base class, the base class name is hidden in the derived class.

Related References
- "Virtual Base Classes" on page 297
- "Incomplete Class Declarations" on page 257
- "C++ Scope Resolution Operator ::" on page 107

Inherited Member Access

This section discusses the access rules affecting a protected nonstatic base class member and how to declare a derived class using an access specifier.

Protected Members

C++ A protected nonstatic base class member can be accessed by members and friends of any classes derived from that base class by using one of the following:
- A pointer to a directly or indirectly derived class
- A reference to a directly or indirectly derived class
- An object of a directly or indirectly derived class

If a class is derived privately from a base class, all protected base class members become private members of the derived class.

If you reference a protected nonstatic member `x` of a base class `A` in a friend or a member function of a derived class `B`, you must access `x` through a pointer to, reference to, or object of a class derived from `A`. However, if you are accessing `x` to create a pointer to member, you must qualify `x` with a nested name specifier that names the derived class `B`. The following example demonstrates this:

```
class A {
public:
protected:
  int i;
};


class B : public A {
  friend void f(A*, B*);
  void g(A*);
};

void f(A* pa, B* pb) {
//  pa->i = 1;
  pb->i = 2;

//  int A::* point_i = &A::i;
  int A::* point_i2 = &B::i;
}

void B::g(A* pa) {
//  pa->i = 1;
  i = 2;

//  int A::* point_i = &A::i;
  int A::* point_i2 = &B::i;
}

void h(A* pa, B* pb) {
//  pa->i = 1;
//  pb->i = 2;
}

int main() { }
```

Class A contains one protected data member, an integer i. Because B derives from A, the members of B have access to the protected member of A. Function f() is a friend of class B:

- The compiler would not allow pa->i = 1 because pa is not a pointer to the derived class B.
- The compiler would not allow int A::* point_i = &A::i because i has not been qualified with the name of the derived class B.

Function g() is a member function of class B. The previous list of remarks about which statements the compiler would and would not allow apply for g() except for the following:

- The compiler allows i = 2 because it is equivalent to this->i = 2.

Function h() cannot access any of the protected members of A because h() is neither a friend or a member of a derived class of A.

Related References

- "References" on page 96
- "Objects" on page 35

Access Control of Base Class Members

C++ When you declare a derived class, an access specifier can precede each base class in the base list of the derived class. This does not alter the access attributes of the individual members of a base class as seen by the base class, but allows the derived class to restrict the access control of the members of a base class.

You can derive classes using any of the three access specifiers:

- In a **public** base class, public and protected members of the base class remain public and protected members of the derived class.
- In a **protected** base class, public and protected members of the base class are protected members of the derived class.
- In a **private** base class, public and protected members of the base class become private members of the derived class.

In all cases, private members of the base class remain private. Private members of the base class cannot be used by the derived class unless friend declarations within the base class explicitly grant access to them.

In the following example, class d is derived publicly from class b. Class b is declared a public base class by this declaration.

```
class b { };
class d : public b // public derivation
{ };
```

You can use both a structure and a class as base classes in the base list of a derived class declaration:

- If the derived class is declared with the keyword **class**, the default access specifier in its base list specifiers is **private**.

- If the derived class is declared with the keyword **struct**, the default access specifier in its base list specifiers is **public**.

In the following example, private derivation is used by default because no access specifier is used in the base list and the derived class is declared with the keyword **class**:

```
struct B
{ };
class D : B // private derivation
{ };
```

Members and friends of a class can implicitly convert a pointer to an object of that class to a pointer to either:

- A direct private base class
- A protected base class (either direct or indirect)

Related References

- “Member Access” on page 276
- “Member Scope” on page 267

The using Declaration and Class Members

C++ A using declaration in a definition of a class A allows you to introduce a *name* of a data member or member function from a base class of A into the scope of A.

You would need a using declaration in a class definition if you want to create a set of overload a member functions from base and derived classes, or you want to change the access of a class member.

Syntax – using Declaration

```
►►──using──┬─┬──────────┬──┬────┬──nested_name_specifier──unqualified_id──;─┬──────────►◄
           │ └─typename─┘  └─::─┘                                           │
           └─::──unqualified_id──;──────────────────────────────────────────┘
```

A using declaration in a class A may name one of the following:

- A member of a base class of A
- A member of an anonymous union that is a member of a base class of A
- An enumerator for an enumeration type that is a member of a base class of A

The following example demonstrates this:

```
struct Z {
  int g();
};

struct A {
  void f();
  enum E { e };
  union { int u; };
};

struct B : A {
  using A::f;
  using A::e;
  using A::u;
//  using Z::g;
};
```

The compiler would not allow the using declaration using Z::g because Z is not a base class of A.

A using declaration cannot name a template. For example, the compiler will not allow the following:

```
struct A {
  template<class T> void f(T);
};

struct B : A {
  using A::f<int>;
};
```

Every instance of the name mentioned in a using declaration must be accessible. The following example demonstrates this:

```
struct A {
private:
  void f(int);
public:
  int f();
protected:
  void g();
};

struct B : A {
//  using A::f;
  using A::g;
};
```

The compiler would not allow the using declaration using A::f because void A::f(int) is not accessible from B even though int A::f() is accessible.

Overloading Member Functions from Base and Derived Classes

C++ A member function named f in a class A will hide all other members named f in the base classes of A, regardless of return types or arguments. The following example demonstrates this:

```
struct A {
  void f() { }
};

struct B : A {
  void f(int) { }
};

int main() {
  B obj_B;
  obj_B.f(3);
//  obj_B.f();
}
```

The compiler would not allow the function call obj_B.f() because the declaration of void B::f(int) has hidden A::f().

To overload, rather than hide, a function of a base class A in a derived class B, you introduce the name of the function into the scope of B with a using declaration. The following example is the same as the previous example except for the using declaration using A::f:

```
struct A {
  void f() { }
};

struct B : A {
  using A::f;
  void f(int) { }
};

int main() {
  B obj_B;
  obj_B.f(3);
  obj_B.f();
}
```

Because of the using declaration in class B, the name f is overloaded with two functions. The compiler will now allow the function call obj_B.f().

You can overload virtual functions in the same way. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct A {
  virtual void f() { cout << "void A::f()" << endl; }
  virtual void f(int) { cout << "void A::f(int)" << endl; }
};

struct B : A {
  using A::f;
  void f(int) { cout << "void B::f(int)" << endl; }
};

int main() {
  B obj_B;
  B* pb = &obj_B;
  pb->f(3);
  pb->f();
}
```

The following is the output of the above example:

```
void B::f(int)
void A::f()
```

Suppose that you introduce a function f from a base class A a derived class B with a using declaration, and there exists a function named B::f that has the same parameter types as A::f. Function B::f will hide, rather than conflict with, function A::f. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct A {
  void f() { }
  void f(int) { cout << "void A::f(int)" << endl; }
};

struct B : A {
  using A::f;
  void f(int) { cout << "void B::f(int)" << endl; }
};
```

```
int main() {
  B obj_B;
  obj_B.f(3);
}
```

The following is the output of the above example:

```
void B::f(int)
```

Changing the Access of a Class Member

C++ Suppose class B is a direct base class of class A. To restrict access of class B to the members of class A, derive B from A using either the access specifiers **protected** or **private**.

To increase the access of a member x of class A inherited from class B, use a using declaration. You cannot restrict the access to x with a using declaration. You may increase the access of the following members:

- A member inherited as **private**. (You cannot increase the access of a member declared as **private** because a using declaration must have access to the member's name.)
- A member either inherited or declared as **protected**

The following example demonstrates this:

```
struct A {
protected:
  int y;
public:
  int z;
};

struct B : private A { };

struct C : private A {
public:
  using A::y;
  using A::z;
};

struct D : private A {
protected:
  using A::y;
  using A::z;
};

struct E : D {
  void f() {
    y = 1;
    z = 2;
  }
};

struct F : A {
public:
  using A::y;
private:
  using A::z;
};

int main() {
  B obj_B;
//  obj_B.y = 3;
//  obj_B.z = 4;

  C obj_C;
  obj_C.y = 5;
  obj_C.z = 6;

  D obj_D;
//  obj_D.y = 7;
//  obj_D.z = 8;

  F obj_F;
  obj_F.y = 9;
  obj_F.z = 10;
}
```

The compiler would not allow the following assignments from the above example:

- `obj_B.y = 3` and `obj_B.z = 4`: Members `y` and `z` have been inherited as **private**.
- `obj_D.y = 7` and `obj_D.z = 8`: Members `y` and `z` have been inherited as **private**, but their access have been changed to **protected**.

The compiler allows the following statements from the above example:

- `y = 1` and `z = 2` in `D::f()`: Members `y` and `z` have been inherited as **private**, but their access have been changed to **protected**.
- `obj_C.y = 5` and `obj_C.z = 6`: Members `y` and `z` have been inherited as **private**, but their access have been changed to **public**.
- `obj_F.y = 9`: The access of member `y` has been changed from **protected** to **public**.
- `obj_F.z = 10`: The access of member `z` is still **public**. The **private** using declaration `using A::z` has no effect on the access of `z`.

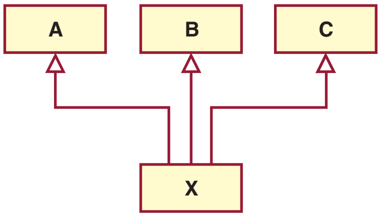
Multiple Inheritance

C++ You can derive a class from any number of base classes. Deriving a class from more than one direct base class is called *multiple inheritance*.

In the following example, classes `A`, `B`, and `C` are direct base classes for the derived class `X`:

```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class X : public A, private B, public C { /* ... */ };
```

The following *inheritance graph* describes the inheritance relationships of the above example. An arrow points to the direct base class of the class at the tail of the arrow:

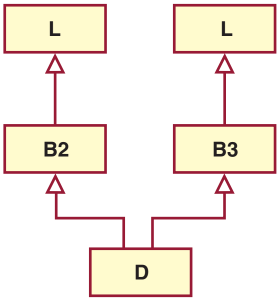


The order of derivation is relevant only to determine the order of default initialization by constructors and cleanup by destructors.

A direct base class cannot appear in the base list of a derived class more than once:

```
class B1 { /* ... */ };                   // direct base class
class D : public B1, private B1 { /* ... */ }; // error
```

However, a derived class can inherit an indirect base class more than once, as shown in the following example:



```
class L { /* ... */ };                  // indirect base class
class B2 : public L { /* ... */ };
class B3 : public L { /* ... */ };
class D : public B2, public B3 { /* ... */ }; // valid
```

In the above example, class D inherits the indirect base class L once through class B2 and once through class B3. However, this may lead to ambiguities because two subobjects of class L exist, and both are accessible through class D. You can avoid this ambiguity by referring to class L using a qualified class name. For example:

```
B2::L
```

or

```
B3::L.
```

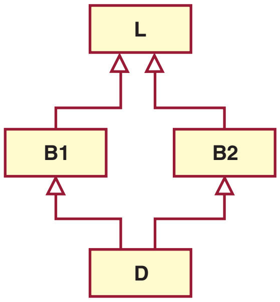
You can also avoid this ambiguity by using the base specifier **virtual** to declare a base class.

Virtual Base Classes

C++ Suppose you have two derived classes B and C that have a common base class A, and you also have another class D that inherits from B and C. You can declare the base class A as *virtual* to ensure that B and C share the same subobject of A.

In the following example, an object of class D has two distinct subobjects of class L, one through class B1 and another through class B2. You can use the keyword **virtual** in front of the base class specifiers in the *base lists* of classes B1 and B2 to indicate that only one subobject of type L, shared by class B1 and class B2, exists.

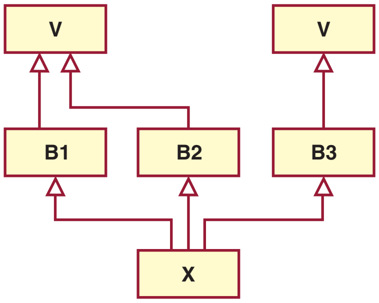
For example:



```
class L { /* ... */ }; // indirect base class
class B1 : virtual public L { /* ... */ };
class B2 : virtual public L { /* ... */ };
class D : public B1, public B2 { /* ... */ }; // valid
```

Using the keyword **virtual** in this example ensures that an object of class D inherits only one subobject of class L.

A derived class can have both virtual and nonvirtual base classes. For example:



```
class V { /* ... */ };
class B1 : virtual public V { /* ... */ };
class B2 : virtual public V { /* ... */ };
class B3 : public V { /* ... */ };
class X : public B1, public B2, public B3 { /* ... */
};
```

In the above example, class X has two subobjects of class V, one that is shared by classes B1 and B2 and one through class B3.

Multiple Access

C++ In an inheritance graph containing virtual base classes, a name that can be reached through more than one path is accessed through the path that gives the most access.

For example:

```
class L {
public:
  void f();
};

class B1 : private virtual L { };

class B2 : public virtual L { };

class D : public B1, public B2 {
public:
  void f() {
    // L::f() is accessed through B2
    // and is public
    L::f();
  }
};
```

In the above example, the function f() is accessed through class B2. Because class B2 is inherited publicly and class B1 is inherited privately, class B2 offers more access.

Ambiguous Base Classes

C++ When you derive classes, ambiguities can result if base and derived classes have members with the same names. Access to a base class member is ambiguous if you use a name or qualified name that does not refer to a unique function or object. The declaration of a member with an ambiguous name in a derived class is not an error. The ambiguity is only flagged as an error if you use the ambiguous member name.

For example, suppose that two classes named A and B both have a member named x, and a class named C inherits from both A and B. An attempt to access x from class C would be ambiguous. You can resolve ambiguity by qualifying a member with its class name using the scope resolution (::) operator.

```
class B1 {
public:
  int i;
  int j;
  void g(int) { }
};

class B2 {
public:
  int j;
  void g() { }
};

class D : public B1, public B2 {
public:
  int i;
};

int main() {
  D dobj;
  D *dptr = &dobj;
  dptr->i = 5;
//  dptr->j = 10;
  dptr->B1::j = 10;
//  dobj.g();
  dobj.B2::g();
}
```

The statement dptr->j = 10 is ambiguous because the name j appears both in B1 and B2. The statement dobj.g() is ambiguous because the name g appears both in B1 and B2, even though B1::g(int) and B2::g() have different parameters.

The compiler checks for ambiguities at compile time. Because ambiguity checking occurs before access control or type checking, ambiguities may result even if only one of several members with the same name is accessible from the derived class.

Name Hiding

Suppose two subobjects named A and B both have a member name x. The member name x of subobject B *hides* the member name x of subobject A if A is a base class of B. The following example demonstrates this:

```
struct A {
   int x;
};

struct B: A {
   int x;
};
```

```
struct C: A, B {
   void f() { x = 0; }
};

int main() {
   C i;
   i.f();
}
```

The assignment `x = 0` in function `C::f()` is not ambiguous because the declaration `B::x` has hidden `A::x`. However, the compiler will warn you that deriving `C` from `A` is redundant because you already have access to the subobject `A` through `B`.

A base class declaration can be hidden along one path in the inheritance graph and not hidden along another path. The following example demonstrates this:

```
struct A { int x; };
struct B { int y; };
struct C: A, virtual B { };
struct D: A, virtual B {
   int x;
   int y;
};
struct E: C, D { };

int main() {
   E e;
//   e.x = 1;
   e.y = 2;
}
```

The assignment `e.x = 1` is ambiguous. The declaration `D::x` hides `A::x` along the path `D::A::x`, but it does not hide `A::x` along the path `D::A::x`. Therefore the variable `x` could refer to either `D::x` or `A::x`. The assignment `e.y = 2` is not ambiguous. The declaration `D::y` hides `B::y` along both paths `D::B::y` and `C::B::y` because `B` is a virtual base class.

Ambiguity and using Declarations

Suppose you have a class named `C` that inherits from a class named `A`, and `x` is a member name of `A`. If you use a using declaration to declare `A::x` in `C`, then `x` is also a member of `C`; `C::x` does not hide `A::x`. Therefore using declarations cannot resolve ambiguities due to inherited members. The following example demonstrates this:

```
struct A {
   int x;
};

struct B: A { };

struct C: A {
   using A::x;
};

struct D: B, C {
   void f() { x = 0; }
};

int main() {
   D i;
   i.f();
}
```

The compiler will not allow the assignment x = 0 in function D::f() because it is ambiguous. The compiler can find x in two ways: as B::x or as C::x.

Unambiguous Class Members

The compiler can unambiguously find static members, nested types, and enumerators defined in a base class A regardless of the number of subobjects of type A an object has. The following example demonstrates this:

```
struct A {
   int x;
   static int s;
   typedef A* Pointer_A;
   enum { e };
};

int A::s;

struct B: A { };

struct C: A { };

struct D: B, C {
   void f() {
      s = 1;
      Pointer_A pa;
      int i = e;
//      x = 1;
   }
};

int main() {
   D i;
   i.f();
}
```

The compiler allows the assignment s = 1, the declaration Pointer_A pa, and the statement int i = e. There is only one static variable s, only one typedef Pointer_A, and only one enumerator e. The compiler would not allow the assignment x = 1 because x can be reached either from class B or class C.

Pointer Conversions

Conversions (either implicit or explicit) from a derived class pointer or reference to a base class pointer or reference must refer unambiguously to the same accessible base class object. (An *accessible base class* is a publicly derived base class that is neither hidden nor ambiguous in the inheritance hierarchy.) For example:

```
class W { /* ... */ };
class X : public W { /* ... */ };
class Y : public W { /* ... */ };
class Z : public X, public Y { /* ... */ };
int main ()
{
      Z z;
      X* xptr = &z;       // valid
      Y* yptr = &z;       // valid
      W* wptr = &z;       // error, ambiguous reference to class W
                          // X's W or Y's W ?
}
```

You can use virtual base classes to avoid ambiguous reference. For example:

```
class W { /* ... */ };
class X : public virtual W { /* ... */ };
class Y : public virtual W { /* ... */ };
class Z : public X, public Y { /* ... */ };
int main ()
{
      Z z;
      X* xptr = &z;      // valid
      Y* yptr = &z;      // valid
      W* wptr = &z;      // valid, W is virtual therefore only one
                         // W subobject exists
}
```

Overload Resolution

Overload resolution takes place *after* the compiler unambiguously finds a given function name. The following example demonstrates this:

```
struct A {
   int f() { return 1; }
};

struct B {
   int f(int arg) { return arg; }
};

struct C: A, B {
   int g() { return f(); }
};
```

The compiler will not allow the function call to `f()` in `C::g()` because the name `f` has been declared both in `A` and `B`. The compiler detects the ambiguity error before overload resolution can select the base match `A::f()`.

Related References
- "C++ Scope Resolution Operator ::" on page 107
- "Virtual Base Classes" on page 297

Virtual Functions

▶ C++ By default, C++ matches a function call with the correct function definition at compile time. This is called *static binding*. You can specify that the compiler match a function call with the correct function definition at run time; this is called *dynamic binding*. You declare a function with the keyword **virtual** if you want the compiler to use dynamic binding for that specific function.

The following examples demonstrate the differences between static and dynamic binding. The first example demonstrates static binding:

```
#include <iostream>
using namespace std;

struct A {
   void f() { cout << "Class A" << endl; }
};

struct B: A {
   void f() { cout << "Class B" << endl; }
};

void g(A& arg) {
   arg.f();
}
```

```
int main() {
   B x;
   g(x);
}
```

The following is the output of the above example:

```
Class A
```

When function `g()` is called, function `A::f()` is called, although the argument refers to an object of type `B`. At compile time, the compiler knows only that the argument of function `g()` will be a reference to an object derived from `A`; it cannot determine whether the argument will be a reference to an object of type `A` or type `B`. However, this can be determined at run time. The following example is the same as the previous example, except that `A::f()` is declared with the **virtual** keyword:

```
#include <iostream>
using namespace std;

struct A {
   virtual void f() { cout << "Class A" << endl; }
};

struct B: A {
   void f() { cout << "Class B" << endl; }
};

void g(A& arg) {
   arg.f();
}

int main() {
   B x;
   g(x);
}
```

The following is the output of the above example:

```
Class B
```

The **virtual** keyword indicates to the compiler that it should choose the appropriate definition of `f()` not by the type of reference, but by the type of object that the reference refers to.

Therefore, a *virtual function* is a member function you may redefine for other derived classes, and can ensure that the compiler will call the redefined virtual function for an object of the corresponding derived class, even if you call that function with a pointer or reference to a base class of the object.

A class that declares or inherits a virtual function is called a *polymorphic class*.

You redefine a virtual member function, like any member function, in any derived class. Suppose you declare a virtual function named `f` in a class `A`, and you derive directly or indirectly from `A` a class named `B`. If you declare a function named `f` in class `B` with the same name and same parameter list as `A::f`, then `B::f` is also virtual (regardless whether or not you declare `B::f` with the **virtual** keyword) and it *overrides* `A::f`. However, if the parameter lists of `A::f` and `B::f` are different, `A::f` and `B::f` are considered different, `B::f` does not override `A::f`, and `B::f` is not virtual (unless you have declared it with the **virtual** keyword). Instead `B::f` *hides* `A::f`. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct A {
   virtual void f() { cout << "Class A" << endl; }
};

struct B: A {
   void f(int) { cout << "Class B" << endl; }
};

struct C: B {
   void f() { cout << "Class C" << endl; }
};

int main() {
   B b; C c;
   A* pa1 = &b;
   A* pa2 = &c;
//   b.f();
   pa1->f();
   pa2->f();
}
```

The following is the output of the above example:

```
Class A
Class C
```

The function `B::f` is not virtual. It hides `A::f`. Thus the compiler will not allow the function call `b.f()`. The function `C::f` is virtual; it overrides `A::f` even though `A::f` is not visible in `C`.

If you declare a base class destructor as virtual, a derived class destructor will override that base class destructor, even though destructors are not inherited.

The return type of an overriding virtual function may differ from the return type of the overridden virtual function. This overriding function would then be called a *covariant virtual function*. Suppose that `B::f` overrides the virtual function `A::f`. The return types of `A::f` and `B::f` may differ if all the following conditions are met:

- The function `B::f` returns a reference or pointer to a class of type `T`, and `A::f` returns a pointer or a reference to an unambiguous direct or indirect base class of `T`.
- The const or volatile qualification of the pointer or reference returned by `B::f` has the same or less const or volatile qualification of the pointer or reference returned by `A::f`.
- The return type of `B::f` must be complete at the point of declaration of `B::f`, or it can be of type `B`.

The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct A { };

class B : private A {
   friend class D;
   friend class F;
};

A global_A;
B global_B;
```

```
struct C {
   virtual A* f() {
      cout << "A* C::f()" << endl;
      return &global_A;
   }
};

struct D : C {
   B* f() {
      cout << "B* D::f()" << endl;
      return &global_B;
   }
};

struct E;

struct F : C {

//   Error:
//   E is incomplete
//   E* f();
};

struct G : C {

//   Error:
//   A is an inaccessible base class of B
//   B* f();
};

int main() {
   D d;
   C* cp = &d;
   D* dp = &d;

   A* ap = cp->f();
   B* bp = dp->f();
};
```

The following is the output of the above example:

```
B* D::f()
B* D::f()
```

The statement `A* ap = cp->f()` calls `D::f()` and converts the pointer returned to type `A*`. The statement `B* bp = dp->f()` calls `D::f()` as well but does not convert the pointer returned; the type returned is `B*`. The compiler would not allow the declaration of the virtual function `F::f()` because `E` is not a complete class. The compiler would not allow the declaration of the virtual function `G::f()` because class `A` is not an accessible base class of `B` (unlike friend classes `D` and `F`, the definition of `B` does not give access to its members for class `G`).

A virtual function cannot be global or static because, by definition, a virtual function is a member function of a base class and relies on a specific object to determine which implementation of the function is called. You can declare a virtual function to be a friend of another class.

If a function is declared virtual in its base class, you can still access it directly using the scope resolution (`::`) operator. In this case, the virtual function call mechanism is suppressed and the function implementation defined in the base class is used. In addition, if you do not override a virtual member function in a derived class, a call to that function uses the function implementation defined in the base class.

A virtual function must be one of the following:
- Defined
- Declared pure
- Defined and declared pure

A base class containing one or more pure virtual member functions is called an *abstract class*.

Ambiguous Virtual Function Calls

C++ You cannot override one virtual function with two or more ambiguous virtual functions. This can happen in a derived class that inherits from two nonvirtual bases that are derived from a virtual base class.

For example:

```
class V {
public:
 virtual void f() { }
};

class A : virtual public V {
  void f() { }
};

class B : virtual public V {
 void f() { }
};

// Error:
// Both A::f() and B::f() try to override V::f()
class D : public A, public B { };

int main() {
  D d;
  V* vptr = &d;

  // which f(), A::f() or B::f()?
 vptr->f();
}
```

The compiler will not allow the definition of class `D`. In class `A`, only `A::f()` will override `V::f()`. Similarly, in class `B`, only `B::f()` will override `V::f()`. However, in class `D`, both `A::f()` and `B::f()` will try to override `V::f()`. This attempt is not allowed because it is not possible to decide which function to call if a `D` object is referenced with a pointer to class `V`, as shown in the above example. Only one function can override a virtual function.

A special case occurs when the ambiguous overriding virtual functions come from separate instances of the same class type. In the following example, class `D` has two separate subobjects of class `A`:

```
#include <iostream>
using namespace std;

struct A {
   virtual void f() { cout << "A::f()" << endl; };
};

struct B : A {
   void f() { cout << "B::f()" << endl;};
};

struct C : A {
   void f() { cout << "C::f()" << endl;};
};

struct D : B, C { };

int main() {
   D d;

   B* bp = &d;
   A* ap = bp;
   D* dp = &d;

   ap->f();
//   dp->f();
}
```

Class D has two occurrences of class A, one inherited from B, and another inherited from C. Therefore there are also two occurrences of the virtual function A::f. The statement ap->f() calls D::B::f. However the compiler would not allow the statement dp->f() because it could either call D::B::f or D::C::f.

Virtual Function Access

▶ C++ The access for a virtual function is specified when it is declared. The access rules for a virtual function are not affected by the access rules for the function that later overrides the virtual function. In general, the access of the overriding member function is not known.

If a virtual function is called with a pointer or reference to a class object, the type of the class object is not used to determine the access of the virtual function. Instead, the type of the pointer or reference to the class object is used.

In the following example, when the function f() is called using a pointer having type B*, bptr is used to determine the access to the function f(). Although the definition of f() defined in class D is executed, the access of the member function f() in class B is used. When the function f() is called using a pointer having type D*, dptr is used to determine the access to the function f(). This call produces an error because f() is declared private in class D.

```
class B {
public:
  virtual void f();
};

class D : public B {
private:
  void f();
};

int main() {
  D dobj;
  B* bptr = &dobj;
  D* dptr = &dobj;

  // valid, virtual B::f() is public,
  // D::f() is called
  bptr->f();

  // error, D::f() is private
  dptr->f();
}
```

Abstract Classes

C++ An *abstract class* is a class that is designed to be specifically used as a base class. An abstract class contains at least one *pure virtual function*. You declare a pure virtual function by using a *pure specifier* (`= 0`) in the declaration of a virtual member function in the class declaration.

The following is an example of an abstract class::

```
class AB {
public:
  virtual void f() = 0;
};
```

Function `AB::f` is a pure virtual function. A function declaration cannot have both a pure specifier and a definition. For example, the compiler will not allow the following:

```
struct A {
  virtual void g() { } = 0;
};
```

You cannot use an abstract class as a parameter type, a function return type, or the type of an explicit conversion, nor can you declare an object of an abstract class. You can, however, declare pointers and references to an abstract class. The following example demonstrates this:

```
struct A {
  virtual void f() = 0;
};

struct B : A {
  virtual void f() { }
};

// Error:
// Class A is an abstract class
// A g();

// Error:
// Class A is an abstract class
// void h(A);
A& i(A&);

int main() {

// Error:
// Class A is an abstract class
//   A a;

   A* pa;
   B b;

// Error:
// Class A is an abstract class
//   static_cast<A>(b);
}
```

Class `A` is an abstract class. The compiler would not allow the function declarations `A g()` or `void h(A)`, declaration of object `a`, nor the static cast of `b` to type `A`.

Virtual member functions are inherited. A class derived from an abstract base class will also be abstract unless you override each pure virtual function in the derived class.

For example:

```
class AB {
public:
  virtual void f() = 0;
};

class D2 : public AB {
  void g();
};

int main() {
  D2 d;
}
```

The compiler will not allow the declaration of object d because D2 is an abstract class; it inherited the pure virtual function f()from AB. The compiler will allow the declaration of object d if you define function D2::g().

Note that you can derive an abstract class from a nonabstract class, and you can override a non-pure virtual function with a pure virtual function.

You can call member functions from a constructor or destructor of an abstract class. However, the results of calling (directly or indirectly) a pure virtual function from its constructor are undefined. The following example demonstrates this:

```
struct A {
  A() {
    direct();
    indirect();
  }
  virtual void direct() = 0;
  virtual void indirect() { direct(); }
};
```

The default constructor of A calls the pure virtual function direct() both directly and indirectly (through indirect()).

The compiler issues a warning for the direct call to the pure virtual function, but not for the indirect call.

Chapter 15. Special Member Functions

C++ The default constructor, destructor, copy constructor, and copy assignment operator are *special member functions*. These functions create, destroy, convert, initialize, and copy class objects.

Constructors and Destructors Overview

C++ Because classes have complicated internal structures, including data and functions, object initialization and cleanup for classes is much more complicated than it is for simple data structures. Constructors and destructors are special member functions of classes that are used to construct and destroy class objects. Construction may involve memory allocation and initialization for objects. Destruction may involve cleanup and deallocation of memory for objects.

Like other member functions, constructors and destructors are declared within a class declaration. They can be defined inline or external to the class declaration. Constructors can have default arguments. Unlike other member functions, constructors can have member initialization lists. The following restrictions apply to constructors and destructors:

- Constructors and destructors do not have return types nor can they return values.
- References and pointers cannot be used on constructors and destructors because their addresses cannot be taken.
- Constructors cannot be declared with the keyword **virtual**.
- Constructors and destructors cannot be declared **static**, **const**, or **volatile**.
- Unions cannot contain class objects that have constructors or destructors.

Constructors and destructors obey the same access rules as member functions. For example, if you declare a constructor with protected access, only derived classes and friends can use it to create class objects.

The compiler automatically calls constructors when defining class objects and calls destructors when class objects go out of scope. A constructor does not allocate memory for the class object its **this** pointer refers to, but may allocate storage for more objects than its class object refers to. If memory allocation is required for objects, constructors can explicitly call the **new** operator. During cleanup, a destructor may release objects allocated by the corresponding constructor. To release objects, use the **delete** operator.

Derived classes do not inherit constructors or destructors from their base classes, but they do call the constructor and destructor of base classes. Destructors can be declared with the keyword **virtual**.

Constructors are also called when local or temporary class objects are created, and destructors are called when local or temporary objects go out of scope.

You can call member functions from constructors or destructors. You can call a virtual function, either directly or indirectly, from a constructor or destructor of a class `A`. In this case, the function called is the one defined in `A` or a base class of `A`, but not a function overridden in any class derived from A. This avoids the possibility of accessing an unconstructed object from a constructor or destructor. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct A {
  virtual void f() { cout << "void A::f()" << endl; }
  virtual void g() { cout << "void A::g()" << endl; }
  virtual void h() { cout << "void A::h()" << endl; }
};

struct B : A {
  virtual void f() { cout << "void B::f()" << endl; }
  B() {
    f();
    g();
    h();
  }
};

struct C : B {
  virtual void f() { cout << "void C::f()" << endl; }
  virtual void g() { cout << "void C::g()" << endl; }
  virtual void h() { cout << "void C::h()" << endl; }
};

int main() {
  C obj;
}
```

The following is the output of the above example:

```
void B::f()
void A::g()
void A::h()
```

The constructor of B does not call any of the functions overridden in C because C has been derived from B, although the example creates an object of type C named obj.

You can use the **typeid** or the **dynamic_cast** operator in constructors or destructors, as well as member initializers of constructors.

Related References
- "The constructor and destructor Function Attributes" on page 166
- "C++ new Operator" on page 126
- "C++ delete Operator" on page 130
- "Free Store" on page 323

Constructors

▶ C++ A *constructor* is a member function with the same name as its class. For example:

```
class X {
public:
  X();      // constructor for class X
};
```

Constructors are used to create, and can initialize, objects of their class type.

You cannot declare a constructor as **virtual** or **static**, nor can you declare a constructor as **const**, **volatile**, or **const volatile**.

You do not specify a return type for a constructor. A return statement in the body of a constructor cannot have a return value.

Related References
- "The constructor and destructor Function Attributes" on page 166
- "Free Store" on page 323

Default Constructors

C++ A *default constructor* is a constructor that either has no parameters, or if it has parameters, *all* the parameters have default values.

If no user-defined constructor exists for a class `A` and one is needed, the compiler implicitly *declares* a constructor `A::A()`. This constructor is an inline public member of its class. The compiler will implicitly *define* `A::A()` when the compiler uses this constructor to create an object of type `A`. The constructor will have no constructor initializer and a null body.

The compiler first implicitly defines the implicitly declared constructors of the base classes and nonstatic data members of a class `A` before defining the implicitly declared constructor of `A`. No default constructor is created for a class that has any constant or reference type members.

A constructor of a class `A` is *trivial* if all the following are true:
- It is implicitly defined
- `A` has no virtual functions and no virtual base classes
- All the direct base classes of `A` have trivial constructors
- The classes of all the nonstatic data members of `A` have trivial constructors

If any of the above are false, then the constructor is *nontrivial*.

A union member cannot be of a class type that has a nontrivial constructor.

Like all functions, a constructor can have default arguments. They are used to initialize member objects. If default values are supplied, the trailing arguments can be omitted in the expression list of the constructor. Note that if a constructor has any arguments that do not have default values, it is not a default constructor.

A *copy constructor* for a class `A` is a constructor whose first parameter is of type `A&`, `const A&`, `volatile A&`, or `const volatile A&`. Copy constructors are used to make a copy of one class object from another class object of the same class type. You cannot use a copy constructor with an argument of the same type as its class; you must use a reference. You can provide copy constructors with additional parameters as long as they all have default arguments. If a user-defined copy constructor does not exist for a class and one is needed, the compiler implicitly creates a copy constructor, with public access, for that class. A copy constructor is not created for a class if any of its members or base classes have an inaccessible copy constructor.

The following code fragment shows two classes with constructors, default constructors, and copy constructors:

```
class X {
public:

  // default constructor, no arguments
  X();

  // constructor
  X(int, int , int = 0);

  // copy constructor
  X(const X&);

  // error, incorrect argument type
  X(X);
};

class Y {
public:

  // default constructor with one
  // default argument
  Y( int = 0);

  // default argument
  // copy constructor
  Y(const Y&, int = 0);
};
```

Related References

- “The constructor and destructor Function Attributes” on page 166
- “Copy Constructors” on page 332

Explicit Initialization with Constructors

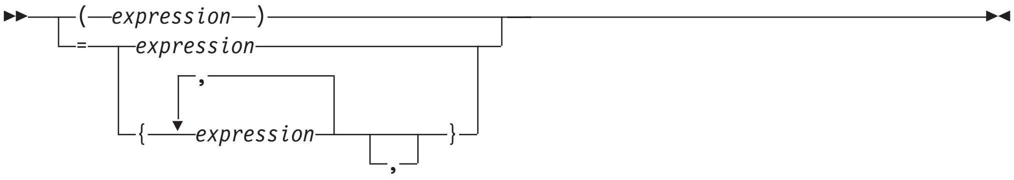
▶ C++ A class object with a constructor must be explicitly initialized or have a default constructor. Except for aggregate initialization, explicit initialization using a constructor is the only way to initialize nonstatic constant and reference class members.

A class object that has no constructors, no virtual functions, no private or protected members, and no base classes is called an *aggregate*. Examples of aggregates are C-style structures and unions.

You explicitly initialize a class object when you create that object. There are two ways to initialize a class object:

- Using a parenthesized expression list. The compiler calls the constructor of the class using this list as the constructor’s argument list.
- Using a single initialization value and the = operator. Because this type of expression is an initialization, not an assignment, the assignment operator function, if one exists, is not called. The type of the single argument must match the type of the first argument to the constructor. If the constructor has remaining arguments, these arguments must have default values.

The syntax for an initializer that explicitly initializes a class object with a constructor is:



The following example shows the declaration and use of several constructors that explicitly initialize class objects:

```
// This example illustrates explicit initialization
// by constructor.
#include <iostream>
using namespace std;

class complx {
  double re, im;
public:

  // default constructor
  complx() : re(0), im(0) { }

  // copy constructor
  complx(const complx& c) { re = c.re; im = c.im; }

  // constructor with default trailing argument
  complx( double r, double i = 0.0) { re = r; im = i; }

  void display() {
    cout << "re = "<< re << " im = " << im << endl;
  }
};

int main() {

  // initialize with complx(double, double)
  complx one(1);

  // initialize with a copy of one
  // using complx::complx(const complx&)
  complx two = one;

  // construct complx(3,4)
  // directly into three
  complx three = complx(3,4);

  // initialize with default constructor
  complx four;

  // complx(double, double) and construct
  // directly into five
  complx five = 5;

  one.display();
  two.display();
  three.display();
  four.display();
  five.display();
}
```

The above example produces the following output:

```
re = 1 im = 0
re = 1 im = 0
re = 3 im = 4
re = 0 im = 0
re = 5 im = 0
```

Related References

- “The constructor and destructor Function Attributes” on page 166
- “Initializers” on page 82

Initializing Base Classes and Members

C++ Constructors can initialize their members in two different ways. A constructor can use the arguments passed to it to initialize member variables in the constructor definition:

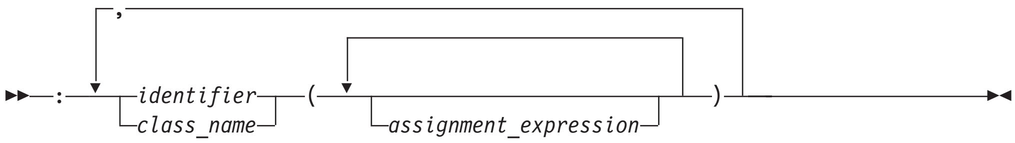
```
complx(double r, double i = 0.0) { re = r; im = i; }
```

Or a constructor can have an *initializer list* within the definition but prior to the function body:

```
complx(double r, double i = 0) : re(r), im(i) { /* ... */ }
```

Both methods assign the argument values to the appropriate data members of the class.

The syntax for a constructor initializer list is:



Include the initialization list as part of the function definition, not as part of the constructor declaration. For example:

```
#include <iostream>
using namespace std;

class B1 {
  int b;
public:
  B1() { cout << "B1::B1()" << endl; };

  // inline constructor
  B1(int i) : b(i) { cout << "B1::B1(int)" << endl; }
};
class B2 {
  int b;
protected:
  B2() { cout << "B2::B2()" << endl; }

  // noninline constructor
  B2(int i);
};

// B2 constructor definition including initialization list
B2::B2(int i) : b(i) { cout << "B2::B2(int)" << endl; }

class D : public B1, public B2 {
  int d1, d2;
public:
  D(int i, int j) : B1(i+1), B2(), d1(i) {
    cout << "D1::D1(int, int)" << endl;
    d2 = j;}
};

int main() {
  D obj(1, 2);
}
```

The following is the output of the above example:

```
B1::B1(int)
B1::B1()
D1::D1(int, int)
```

If you do not explicitly initialize a base class or member that has constructors by calling a constructor, the compiler automatically initializes the base class or member with a default constructor. In the above example, if you leave out the call B2() in the constructor of class D (as shown below), a constructor initializer with an empty expression list is automatically created to initialize B2. The constructors for class D, shown above and below, result in the same construction of an object of class D:

```
class D : public B1, public B2 {
  int d1, d2;
public:

  // call B2() generated by compiler
  D(int i, int j) : B1(i+1), d1(i) {
    cout << "D1::D1(int, int)" << endl;
    d2 = j;}
};
```

In the above example, the compiler will automatically call the default constructor for B2().

Note that you must declare constructors as public or protected to enable a derived class to call them. For example:

```
class B {
  B() { }
};

class D : public B {

  // error: implicit call to private B() not allowed
  D() { }
};
```

The compiler does not allow the definition of D::D() because this constructor cannot access the private constructor B::B().

You must initialize the following with an initializer list: base classes with no default constructors, reference data members, non-static const data members, or a class type which contains a constant data member. The following example demonstrates this:

```
class A {
public:
  A(int) { }
};

class B : public A {
  static const int i;
  const int j;
  int &k;
public:
  B(int& arg) : A(0), j(1), k(arg) { }
};

int main() {
  int x = 0;
  B obj(x);
};
```

The data members `j` and `k`, as well as the base class `A` must be initialized in the initializer list of the constructor of `B`.

You can use data members when initializing members of a class. The following example demonstrate this:

```
struct A {
  int k;
  A(int i) : k(i) { }
};
struct B: A {
  int x;
  int i;
  int j;
  int& r;
  B(int i): r(x), A(i), j(this->i), i(i) { }
};
```

The constructor `B(int i)` initializes the following:

- `B::r` to refer to `B::x`
- Class `A` with the value of the argument to `B(int i)`
- `B::j` with the value of `B::i`
- `B::i` with the value of the argument to `B(int i)`

You can also call member functions (including virtual member functions) or use the operators **typeid** or **dynamic_cast** when initializing members of a class. However if you perform any of these operations in a member initialization list before all base classes have been initialized, the behavior is undefined. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct A {
  int i;
  A(int arg) : i(arg) {
    cout << "Value of i: " << i << endl;
  }
};

struct B : A {
  int j;
  int f() { return i; }
  B();
};

B::B() : A(f()), j(1234) {
   cout << "Value of j: " << j << endl;
}

int main() {
  B obj;
}
```

The output of the above example would be similar to the following:

```
Value of i: 8
Value of j: 1234
```

The behavior of the initializer `A(f())` in the constructor of `B` is undefined. The run time will call `B::f()` and try to access `A::i` even though the base `A` has not been initialized.

The following example is the same as the previous example except that the initializers of `B::B()` have different arguments:

```
#include <iostream>
using namespace std;

struct A {
  int i;
  A(int arg) : i(arg) {
    cout << "Value of i: " << i << endl;
  }
};

struct B : A {
  int j;
  int f() { return i; }
  B();
};

B::B() : A(5678), j(f()) {
   cout << "Value of j: " << j << endl;
}

int main() {
  B obj;
}
```

The following is the output of the above example:

```
Value of i: 5678
Value of j: 5678
```

The behavior of the initializer `j(f())` in the constructor of `B` is well-defined. The base class `A` is already initialized when `B::j` is initialized.

Related References
- "Default Constructors" on page 313
- "The typeid Operator" on page 112
- "dynamic_cast Operator" on page 117

Construction Order of Derived Class Objects

C++ When a derived class object is created using constructors, it is created in the following order:

1. Virtual base classes are initialized, in the order they appear in the base list.
2. Nonvirtual base classes are initialized, in declaration order.
3. Class members are initialized in declaration order (regardless of their order in the initialization list).
4. The body of the constructor is executed.

The following example demonstrates this:

```
#include <iostream>
using namespace std;
struct V {
  V() { cout << "V()" << endl; }
};
struct V2 {
  V2() { cout << "V2()" << endl; }
};
struct A {
  A() { cout << "A()" << endl; }
};
struct B : virtual V {
  B() { cout << "B()" << endl; }
};
struct C : B, virtual V2 {
  C() { cout << "C()" << endl; }
};
struct D : C, virtual V {
  A obj_A;
  D() { cout << "D()" << endl; }
};
int main() {
  D c;
}
```

The following is the output of the above example:

```
V()
V2()
B()
C()
A()
D()
```

The above output lists the order in which the C++ run time calls the constructors to create an object of type D.

Related References

- "Virtual Base Classes" on page 297

Destructors

▶ C++ *Destructors* are usually used to deallocate memory and do other cleanup for a class object and its class members when the object is destroyed. A destructor is called for a class object when that object passes out of scope or is explicitly deleted.

A destructor is a member function with the same name as its class prefixed by a ~ (tilde). For example:

```
class X {
public:
  // Constructor for class X
  X();
  // Destructor for class X
  ~X();
};
```

A destructor takes no arguments and has no return type. Its address cannot be taken. Destructors cannot be declared **const**, **volatile**, **const volatile** or **static**. A destructor can be declared **virtual** or pure **virtual**.

If no user-defined destructor exists for a class and one is needed, the compiler implicitly declares a destructor. This implicitly declared destructor is an inline public member of its class.

The compiler will implicitly define an implicitly declared destructor when the compiler uses the destructor to destroy an object of the destructor's class type. Suppose a class A has an implicitly declared destructor. The following is equivalent to the function the compiler would implicitly define for A:

```
  A::~A() { }
```

The compiler first implicitly defines the implicitly declared destructors of the base classes and nonstatic data members of a class A before defining the implicitly declared destructor of A

A destructor of a class A is *trivial* if all the following are true:
- It is implicitly defined
- All the direct base classes of A have trivial destructors
- The classes of all the nonstatic data members of A have trivial destructors

If any of the above are false, then the destructor is *nontrivial*.

A union member cannot be of a class type that has a nontrivial destructor.

Class members that are class types can have their own destructors. Both base and derived classes can have destructors, although destructors are not inherited. If a base class A or a member of A has a destructor, and a class derived from A does not declare a destructor, a default destructor is generated.

The default destructor calls the destructors of the base class and members of the derived class.

The destructors of base classes and members are called in the reverse order of the completion of their constructor:

1. The destructor for a class object is called before destructors for members and bases are called.
2. Destructors for nonstatic members are called before destructors for base classes are called.
3. Destructors for nonvirtual base classes are called before destructors for virtual base classes are called.

When an exception is thrown for a class object with a destructor, the destructor for the temporary object thrown is not called until control passes out of the catch block.

Destructors are implicitly called when an automatic object (a local object that has been declared **auto** or **register**, or not declared as **static** or **extern**) or temporary object passes out of scope. They are implicitly called at program termination for constructed external and static objects. Destructors are invoked when you use the **delete** operator for objects created with the **new** operator.

For example:

```
#include <string>

class Y {
private:
  char * string;
  int number;
public:
  // Constructor
  Y(const char*, int);
  // Destructor
  ~Y() { delete[] string; }
};

// Define class Y constructor
Y::Y(const char* n, int a) {
  string = strcpy(new char[strlen(n) + 1 ], n);
  number = a;
}

int main () {
  // Create and initialize
  // object of class Y
  Y yobj = Y("somestring", 10);

  // ...

  // Destructor ~Y is called before
  // control returns from main()
}
```

You can use a destructor explicitly to destroy objects, although this practice is not recommended. However to destroy an object created with the placement **new** operator, you can explicitly call the object's destructor. The following example demonstrates this:

```
#include <new>
#include <iostream>
using namespace std;
class A {
  public:
    A() { cout << "A::A()" << endl; }
    ~A() { cout << "A::~A()" << endl; }
};
int main () {
  char* p = new char[sizeof(A)];
  A* ap = new (p) A;
  ap->A::~A();
  delete [] p;
}
```

The statement `A* ap = new (p) A` dynamically creates a new object of type `A` not in the free store but in the memory allocated by `p`. The statement `delete [] p` will delete the storage allocated by `p`, but the run time will still believe that the object pointed to by `ap` still exists until you explicitly call the destructor of `A` (with the statement `ap->A::~A()`).

Nonclass types have a *pseudo destructor*. The following example calls the pseudo destructor for an integer type:

```
typedef int I;
int main() {
  I x = 10;
  x.I::~I();
  x = 20;
}
```

The call to the pseudo destructor, `x.I::~I()`, has no effect at all. Object `x` has not been destroyed; the assignment `x = 20` is still valid. Because pseudo destructors require the syntax for explicitly calling a destructor for a nonclass type to be valid, you can write code without having to know whether or not a destructor exists for a given type.

Related References

- "The constructor and destructor Function Attributes" on page 166
- "Temporary Objects" on page 327

Free Store

C++ *Free store* is a pool of memory available for you to allocate (and deallocate) storage for objects during the execution of your program. The **new** and **delete** operators are used to allocate and deallocate free store, respectively.

You can define your own versions of **new** and **delete** for a class by overloading them. You can declare the **new** and **delete** operators with additional parameters. When **new** and **delete** operate on class objects, the class member operator functions **new** and **delete** are called, if they have been declared.

If you create a class object with the **new** operator, one of the operator functions **operator new()** or **operator new[]()** (if they have been declared) is called to create the object. An **operator new()** or **operator new[]()** for a class is always a static class member, even if it is not declared with the keyword **static**. It has a return type **void*** and its first parameter must be the size of the object type and have type **std::size_t**. It cannot be virtual.

Type **std::size_t** is an implementation-dependent unsigned integral type defined in the standard library header `<cstddef>`. When you overload the **new** operator, you must declare it as a class member, returning type **void***, with its first parameter of type **std::size_t**. You can declare additional parameters in the declaration of **operator new()** or **operator new[]()**. Use the placement syntax to specify values for these parameters in an allocation expression.

The following example overloads two operator **new** functions:

- `X::operator new(size_t sz)`: This overloads the default **new** operator by allocating memory with the C function `malloc()`, and throwing a string (instead of `std::bad_alloc`) if `malloc()` fails.
- `X::operator new(size_t sz, int location)`: This function takes an additional integer parameter, `location`. This function implements a very simplistic "memory manager" that manages the storage of up to three `X` objects.

  Static array `X::buffer` holds three `Node` objects. Each `Node` object contains a pointer to an `X` object named `data` and a boolean variable named `filled`. Each `X` object stores an integer called number.

  When you use this **new** operator, you pass the argument `location` which indicates the array location of `buffer` where you want to "create" your new `X` object. If the array location is not "filled" (the data member of `filled` is equal to `false` at that array location), the **new** operator returns a pointer pointing to the `X` object located at `buffer[location]`.

```
#include <new>
#include <iostream>

using namespace std;

class X;

struct Node {
  X* data;
  bool filled;
  Node() : filled(false) { }
};

class X {
  static Node buffer[];

public:

  int number;

  enum { size = 3};

  void* operator new(size_t sz) throw (const char*) {
    void* p = malloc(sz);
    if (sz == 0) throw "Error: malloc() failed";
    cout << "X::operator new(size_t)" << endl;
    return p;
  }

  void *operator new(size_t sz, int location) throw (const char*) {
    cout << "X::operator new(size_t, " << location << ")" << endl;
    void* p = 0;
    if (location < 0 || location >= size || buffer[location].filled == true) {
      throw "Error: buffer location occupied";
    }
    else {
    p = malloc(sizeof(X));
   if (p == 0) throw "Error: Creating X object failed";
   buffer[location].filled = true;
      buffer[location].data = (X*) p;
    }
    return p;
  }

  static void printbuffer() {
    for (int i = 0; i < size; i++) {
      cout << buffer[i].data->number << endl;
    }
  }

};

Node X::buffer[size];

int main() {
  try {
    X* ptr1 = new X;
    X* ptr2 = new(0) X;
    X* ptr3 = new(1) X;
    X* ptr4 = new(2) X;
    ptr2->number = 10000;
    ptr3->number = 10001;
    ptr4->number = 10002;
    X::printbuffer();
    X* ptr5 = new(0) X;
  }
  catch (const char* message) {
    cout << message << endl;
  }
}
```

The following is the output of the above example:

```
X::operator new(size_t)
X::operator new(size_t, 0)
X::operator new(size_t, 1)
X::operator new(size_t, 2)
10000
10001
10002
X::operator new(size_t, 0)
Error: buffer location occupied
```

The statement `X* ptr1 = new X` calls `X::operator new(sizeof(X))`. The statement `X* ptr2 = new(0) X` calls `X::operator new(sizeof(X),0)`.

The **delete** operator destroys an object created by the **new** operator. The operand of **delete** must be a pointer returned by **new**. If **delete** is called for an object with a destructor, the destructor is invoked before the object is deallocated.

If you destroy a class object with the **delete** operator, the operator function **operator delete()** or **operator delete[]()** (if they have been declared) is called to destroy the object. An **operator delete()** or **operator delete[]()** for a class is always a static member, even if it is not declared with the keyword **static**. Its first parameter must have type **void***. Because **operator delete()** and **operator delete[]()** have a return type **void**, they cannot return a value.

The following example shows the declaration and use of the operator functions **operator new()** and **operator delete()**:

```
#include <cstdlib>
#include <iostream>
using namespace std;

class X {
public:
  void* operator new(size_t sz) throw (const char*) {
    void* p = malloc(sz);
    if (p == 0) throw "malloc() failed";
    return p;
  }

  // single argument
  void operator delete(void* p) {
 cout << "X::operator delete(void*)" << endl;
    free(p);
  }

};

class Y {
  int filler[100];
public:

  // two arguments
  void operator delete(void* p, size_t sz) throw (const char*) {
    cout << "Freeing " << sz << " byte(s)" << endl;
    free(p);
  };

};

int main() {
  X* ptr = new X;

  // call X::operator delete(void*)
  delete ptr;
  Y* yptr = new Y;

  // call Y::operator delete(void*, size_t)
  // with size of Y as second argument
  delete yptr;
}
```

The above example will generate output similar to the following:

```
X::operator delete(void*)
Freeing 400 byte(s)
```

The statement `delete ptr` calls `X::operator delete(void*)`. The statement delete `yptr` calls `Y::operator delete(void*, size_t)`.

The result of trying to access a deleted object is undefined because the value of the object can change after deletion.

If **new** and **delete** are called for a class object that does not declare the operator functions **new** and **delete**, or they are called for a nonclass object, the global operators **new** and **delete** are used. The global operators **new** and **delete** are provided in the C++ library.

The C++ operators for allocating and deallocating arrays of class objects are **operator new[ ]()** and **operator delete[ ]()**.

You cannot declare the **delete** operator as virtual. However you can add polymorphic behavior to your **delete** operators by declaring the destructor of a base class as virtual. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct A {
  virtual ~A() { cout << "~A()" << endl; };
  void operator delete(void* p) {
    cout << "A::operator delete" << endl;
    free(p);
  }
};

struct B : A {
  void operator delete(void* p) {
    cout << "B::operator delete" << endl;
    free(p);
  }
};

int main() {
  A* ap = new B;
  delete ap;
}
```

The following is the output of the above example:

```
~A()
B::operator delete
```

The statement `delete ap` uses the **delete** operator from class `B` instead of class `A` because the destructor of `A` has been declared as virtual.

Although you can get polymorphic behavior from the **delete** operator, the **delete** operator that is statically visible must still be accessible even though another **delete** operator might be called. For example, in the above example, the function `A::operator delete(void*)` must be accessible even though the example calls `B::operator delete(void*)` instead.

Virtual destructors do not have any affect on deallocation operators for arrays (**operator delete[]()**). The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct A {
  virtual ~A() { cout << "~A()" << endl; }
  void operator delete[](void* p, size_t) {
    cout << "A::operator delete[]" << endl;
    ::delete [] p;
  }
};

struct B : A {
  void operator delete[](void* p, size_t) {
    cout << "B::operator delete[]" << endl;
    ::delete [] p;
  }
};

int main() {
  A* bp = new B[3];
  delete[] bp;
};
```

The behavior of the statement `delete[] bp` is undefined.

When you overload the **delete** operator, you must declare it as class member, returning type **void**, with the first parameter having type **void***, as described above. You can add a second parameter of type **size_t** to the declaration. You can only have one **operator delete()** or **operator delete[]()** for a single class.

Related References
- "C++ new Operator" on page 126
- "C++ delete Operator" on page 130
- "Placement Syntax" on page 128
- "Allocation and Deallocation Functions" on page 184

Temporary Objects

C++ It is sometimes necessary for the compiler to create temporary objects. They are used during reference initialization and during evaluation of expressions including standard type conversions, argument passing, function returns, and evaluation of the **throw** expression.

When a temporary object is created to initialize a reference variable, the name of the temporary object has the same scope as that of the reference variable. When a temporary object is created during the evaluation of a full-expression (an expression that is not a subexpression of another expression), it is destroyed as the last step in its evaluation that lexically contains the point where it was created.

There are two exceptions in the destruction of full-expressions:

- The expression appears as an initializer for a declaration defining an object: the temporary object is destroyed when the initialization is complete.
- A reference is bound to a temporary object: the temporary object is destroyed at the end of the reference's lifetime.

If a temporary object is created for a class with constructors, the compiler calls the appropriate (matching) constructor to create the temporary object.

When a temporary object is destroyed and a destructor exists, the compiler calls the destructor to destroy the temporary object. When you exit from the scope in which the temporary object was created, it is destroyed. If a reference is bound to a temporary object, the temporary object is destroyed when the reference passes out of scope unless it is destroyed earlier by a break in the flow of control. For example, a temporary object created by a constructor initializer for a reference member is destroyed on leaving the constructor.

In cases where such temporary objects are redundant, the compiler does not construct them, in order to create more efficient optimized code. This behavior could be a consideration when you are debugging your programs, especially for memory problems.

Related References
- "Arguments of catch Blocks" on page 375
- "Initializing References" on page 97
- "Cast Expressions" on page 131
- "Function Return Values" on page 183

User-Defined Conversions

C++ *User-defined conversions* allow you to specify object conversions with constructors or with conversion functions. User-defined conversions are implicitly used in addition to standard conversions for conversion of initializers, functions arguments, function return values, expression operands, expressions controlling iteration, selection statements, and explicit type conversions.

There are two types of user-defined conversions:
- Conversion by constructor
- Conversion functions

The compiler can use only one user-defined conversion (either a conversion constructor or a conversion function) when implicitly converting a single value. The following example demonstrates this:

```
class A {
  int x;
public:
  operator int() { return x; };
};

class B {
  A y;
public:
  operator A() { return y; };
};

int main () {
  B b_obj;
//  int i = b_obj;
  int j = A(b_obj);
}
```

The compiler would not allow the statement int i = b_obj. The compiler would have to implicitly convert b_obj into an object of type A (with B::operator A()), then implicitly convert that object to an integer (with A::operator int()). The statement int j = A(b_obj) explicitly converts b_obj into an object of type A, then implicitly converts that object to an integer.

User-defined conversions must be unambiguous, or they are not called. A conversion function in a derived class does not hide another conversion function in a base class unless both conversion functions convert to the same type. Function overload resolution selects the most appropriate conversion function. The following example demonstrates this:

```
class A {
  int a_int;
  char* a_carp;
public:
  operator int() { return a_int; }
  operator char*() { return a_carp; }
};

class B : public A {
  float b_float;
  char* b_carp;
public:
  operator float() { return b_float; }
  operator char*() { return b_carp; }
};

int main () {
  B b_obj;
//  long a = b_obj;
  char* c_p = b_obj;
}
```

The compiler would not allow the statement long a = b_obj. The compiler could either use A::operator int() or B::operator float() to convert b_obj into a **long**. The statement char* c_p = b_obj uses B::operator char*() to convert b_obj into a **char*** because B::operator char*() hides A::operator char*().

When you call a constructor with an argument and you have not defined a constructor accepting that argument type, only standard conversions are used to convert the argument to another argument type acceptable to a constructor for that class. No other constructors or conversions functions are called to convert the argument to a type acceptable to a constructor defined for that class. The following example demonstrates this:

```
class A {
public:
  A() { }
  A(int) { }
};

int main() {
   A a1 = 1.234;
//   A moocow = "text string";
}
```

The compiler allows the statement `A a1 = 1.234`. The compiler uses the standard conversion of converting `1.234` into an **int**, then implicitly calls the converting constructor `A(int)`. The compiler would not allow the statement `A moocow = "text string"`; converting a text string to an integer is not a standard conversion.

Related References
- "Standard Type Conversions" on page 150

Conversion by Constructor

C++ A *converting constructor* is a single-parameter constructor that is declared without the function specifier **explicit**. The compiler uses converting constructors to convert objects from the type of the first parameter to the type of the converting constructor's class. The following example demonstrates this:

```
class Y {
  int a, b;
  char* name;
public:
  Y(int i) { };
  Y(const char* n, int j = 0) { };
};

void add(Y) { };

int main() {

  // equivalent to
  // obj1 = Y(2)
  Y obj1 = 2;

  // equivalent to
  // obj2 = Y("somestring",0)
  Y obj2 = "somestring";

  // equivalent to
  // obj1 = Y(10)
  obj1 = 10;

  // equivalent to
  // add(Y(5))
  add(5);
}
```

The above example has the following two converting constructors:
- `Y(int i)`which is used to convert integers to objects of class Y.
- `Y(const char* n, int j = 0)` which is used to convert pointers to strings to objects of class Y.

The compiler will not implicitly convert types as demonstrated above with constructors declared with the **explicit** keyword. The compiler will only use explicitly declared constructors in **new** expressions, the **static_cast** expressions and explicit casts, and the initialization of bases and members. The following example demonstrates this:

```
class A {
public:
  explicit A() { };
  explicit A(int) { };
};

int main() {
  A z;
//  A y = 1;
```

```
  A x = A(1);
  A w(1);
  A* v = new A(1);
  A u = (A)1;
  A t = static_cast<A>(1);
}
```

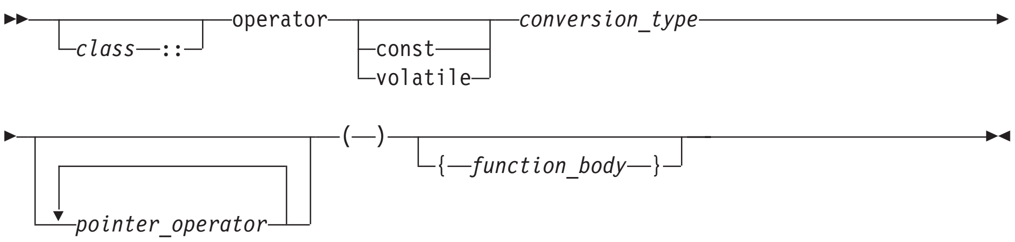
The compiler would not allow the statement `A y = 1` because this is an implicit conversion; class `A` has no conversion constructors.

A copy constructor is a converting constructor.

Related References
- "The explicit Keyword" on page 156
- "C++ new Operator" on page 126
- "static_cast Operator" on page 114

Conversion Functions

C++ You can define a member function of a class, called a *conversion function*, that converts from the type of its class to another specified type.



A conversion function that belongs to a class `X` specifies a conversion from the class type `X` to the type specified by the *conversion_type*. The following code fragment shows a conversion function called `operator int()`:

```
class Y {
  int b;
public:
  operator int();
};
Y::operator int() {
  return b;
}
void f(Y obj) {
  int i = int(obj);
  int j = (int)obj;
  int k = i + obj;
}
```

All three statements in function `f(Y)` use the conversion function `Y::operator int()`.

Classes, enumerations, **typedef** names, function types, or array types cannot be declared or defined in the *conversion_type*. You cannot use a conversion function to convert an object of type `A` to type `A`, to a base class of `A`, or to **void**.

Conversion functions have no arguments, and the return type is implicitly the conversion type. Conversion functions can be inherited. You can have virtual conversion functions but not static ones.

Related References

- “Standard Type Conversions” on page 150
- “User-Defined Conversions” on page 328
- “Conversion by Constructor” on page 330
- Chapter 6, “Implicit Type Conversions,” on page 149

Copy Constructors

C++ The *copy constructor* lets you create a new object from an existing one by initialization. A copy constructor of a class `A` is a non-template constructor in which the first parameter is of type `A&`, `const A&`, `volatile A&`, or `const volatile A&`, and the rest of its parameters (if there are any) have default values.

If you do not declare a copy constructor for a class `A`, the compiler will implicitly declare one for you, which will be an inline public member.

The following example demonstrates implicitly defined and user-defined copy constructors:

```
#include <iostream>
using namespace std;

struct A {
  int i;
  A() : i(10) { }
};

struct B {
  int j;
  B() : j(20) {
    cout << "Constructor B(), j = " << j << endl;
  }

  B(B& arg) : j(arg.j) {
    cout << "Copy constructor B(B&), j = " << j << endl;
  }

  B(const B&, int val = 30) : j(val) {
    cout << "Copy constructor B(const B&, int), j = " << j << endl;
  }
};

struct C {
  C() { }
  C(C&) { }
};

int main() {
  A a;
  A a1(a);
  B b;
  const B b_const;
  B b1(b);
  B b2(b_const);
  const C c_const;
//  C c1(c_const);
}
```

The following is the output of the above example:

```
Constructor B(), j = 20
Constructor B(), j = 20
Copy constructor B(B&), j = 20
Copy constructor B(const B&, int), j = 30
```

The statement `A a1(a)` creates a new object from `a` with an implicitly defined copy constructor. The statement `B b1(b)` creates a new object from `b` with the user-defined copy constructor `B::B(B&)`. The statement `B b2(b_const)` creates a new object with the copy constructor `B::B(const B&, int)`. The compiler would not allow the statement `C c1(c_const)` because a copy constructor that takes as its first parameter an object of type `const C&` has not been defined.

The implicitly declared copy constructor of a class `A` will have the form `A::A(const A&)` if the following are true:

- The direct and virtual bases of `A` have copy constructors whose first parameters have been qualified with **const** or **const volatile**
- The nonstatic class type or array of class type data members of `A` have copy constructors whose first parameters have been qualified with **const** or **const volatile**

If the above are not true for a class `A`, the compiler will implicitly declare a copy constructor with the form `A::A(A&)`.

The compiler cannot allow a program in which the compiler must implicitly define a copy constructor for a class `A` and one or more of the following are true:

- Class `A` has a nonstatic data member of a type which has an inaccessible or ambiguous copy constructor.
- Class `A` is derived from a class which has an inaccessible or ambiguous copy constructor.

The compiler will implicitly define an implicitly declared constructor of a class `A` if you initialize an object of type `A` or an object derived from class `A`.

An implicitly defined copy constructor will copy the bases and members of an object in the same order that a constructor would initialize the bases and members of the object.

Related References

- "Constructors and Destructors Overview" on page 311

Copy Assignment Operators

C++ The *copy assignment operator* lets you create a new object from an existing one by initialization. A copy assignment operator of a class `A` is a nonstatic non-template member function that has one of the following forms:

- `A::operator=(A)`
- `A::operator=(A&)`
- `A::operator=(const A&)`
- `A::operator=(volatile A&)`
- `A::operator=(const volatile A&)`

If you do not declare a copy assignment operator for a class `A`, the compiler will implicitly declare one for you which will be inline public.

The following example demonstrates implicitly defined and user-defined copy assignment operators:

```
#include <iostream>
using namespace std;

struct A {
  A& operator=(const A&) {
    cout << "A::operator=(const A&)" << endl;
    return *this;
  }

  A& operator=(A&) {
    cout << "A::operator=(A&)" << endl;
    return *this;
  }

};
class B {
  A a;
};

struct C {
  C& operator=(C&) {
    cout << "C::operator=(C&)" << endl;
    return *this;
  }
  C() { }
};

int main() {
  B x, y;
  x = y;

  A w, z;
  w = z;

  C i;
  const C j();
//  i = j;
}
```

The following is the output of the above example:

```
A::operator=(const A&)
A::operator=(A&)
```

The assignment x = y calls the implicitly defined copy assignment operator of B, which calls the user-defined copy assignment operator A::operator=(const A&). The assignment w = z calls the user-defined operator A::operator=(A&). The compiler will not allow the assignment i = j because an operator C::operator=(const C&) has not been defined.

The implicitly declared copy assignment operator of a class A will have the form A& A::operator=(const A&) if the following are true:

- A direct or virtual base B of class A has a copy assignment operator whose parameter is of type const B&, const volatile B&, or B.
- A non-static class type data member of type X that belongs to class A has a copy constructor whose parameter is of type const X&, const volatile X&, or X.

If the above are not true for a class A, the compiler will implicitly declare a copy assignment operator with the form A& A::operator=(A&).

The implicitly declared copy assignment operator returns a reference to the operator's argument.

The copy assignment operator of a derived class hides the copy assignment operator of its base class.

The compiler cannot allow a program in which the compiler must implicitly define a copy assignment operator for a class `A` and one or more of the following are true:

- Class `A` has a nonstatic data member of a **const** type or a reference type
- Class `A` has a nonstatic data member of a type which has an inaccessible copy assignment operator
- Class `A` is derived from a base class with an inaccessible copy assignment operator.

An implicitly defined copy assignment operator of a class `A` will first assign the direct base classes of `A` in the order that they appear in the definition of `A`. Next, the implicitly defined copy assignment operator will assign the nonstatic data members of `A` in the order of their declaration in the definition of `A`.

Related References

- "Assignment Expressions" on page 144

Chapter 16. Templates

C++ A *template* describes a set of related classes or set of related functions in which a list of parameters in the declaration describe how the members of the set vary. The compiler generates new classes or functions when you supply arguments for these parameters; this process is called *template instantiation*. This class or function definition generated from a template and a set of template parameters is called a *specialization*.

Syntax – Template Declaration

```
►►──┬──────────┬──template──<──template_parameter_list──>──declaration──────────►◄
    └─export───┘
```

The compiler accepts and silently ignores the export keyword on a template.

The *template_parameter_list* is a comma-separated list of the following kinds of template parameters:

- non-type
- type
- template

The *declaration* is one of the following::

- a declaration or definition of a function or a class
- a definition of a member function or a member class of a class template
- a definition of a static data member of a class template
- a definition of a static data member of a class nested within a class template
- a definition of a member template of a class or class template

The *identifier* of a *type* is defined to be a *type_name* in the scope of the template declaration. A template declaration can appear as a namespace scope or class scope declaration.

The following example demonstrates the use of a class template:

```
template<class L> class Key
{
    L k;
    L* kptr;
    int length;
public:
    Key(L);
    // ...
};
```

Suppose the following declarations appear later:

```
Key<int> i;
Key<char*> c;
Key<mytype> m;
```

The compiler would create three objects. The following table shows the definitions of these three objects if they were written out in source form as regular classes, not as templates:

class Key<int> i;	class Key<char*> c;	class Key<mytype> m;

class Key
{
    int k;
    int * kptr;
    int length;
public:
    Key(int);
    // ...
};
	class Key
	{
	    char* k;
	    char** kptr;
	    int length;
	public:
	    Key(char*);
	    // ...
	};
		class Key
		{
		    mytype k;
		    mytype* kptr;
		    int length;
		public:
		    Key(mytype);
		    // ...
		};

Note that these three classes have different names. The arguments contained within the angle braces are not just the arguments to the class names, but part of the class names themselves. `Key<int>` and `Key<char*>` are class names.

Template Parameters

C++ There are three kinds of template parameters:

- type
- non-type
- template

You can interchange the keywords **class** and **typename** in a template parameter declaration. You cannot use storage class specifiers (**static** and **auto**) in a template parameter declaration.

Type Template Parameters

C++ The following is the syntax of a type template parameter declaration:

Syntax – Type Template Parameter Declaration

```
►►──┬─class────┬──identifier──┬──────────┬──────────────────────────────►◄
    └─typename─┘              └─=──type──┘
```

The *identifier* is the name of a type.

Non-Type Template Parameters

C++ The syntax of a non-type template parameter is the same as a declaration of one of the following types:

- integral or enumeration
- pointer to object or pointer to function
- reference to object or reference to function
- pointer to member

Non-type template parameters that are declared as arrays or functions are converted to pointers or pointer to functions, respectively. The following example demonstrates this:

```
template<int a[4]> struct A { };
template<int f(int)> struct B { };

int i;
int g(int) { return 0;}

A<&i> x;
B<&g> y;
```

The type of `&i` is **int ***, and the type of `&g` is **int (*)(int)**.

You may qualify a non-type template parameter with **const** or **volatile**.

You cannot declare a non-type template parameter as a floating point, class, or void type.

Non-type template parameters are not lvalues.

Template Template Parameters

C++ The following is the syntax of a template template parameter declaration:

Syntax – Template Template Parameter Declaration

```
►►──template──<──template-parameter-list──>──class──┬────────────┬──┬─────────────────┬────────►◄
                                                    └─identifier─┘  └─=──id-expression─┘
```

The following example demonstrates a declaration and use of a template template parameter:

```
template<template <class T> class X> class A { };
template<class T> class B { };

A<B> a;
```

Default Arguments for Template Parameters

C++ Template parameters may have default arguments. The set of default template arguments accumulates over all declarations of a given template. The following example demonstrates this:

```
template<class T, class U = int> class A;
template<class T = float, class U> class A;

template<class T, class U> class A {
   public:
      T x;
      U y;
};

A<> a;
```

The type of member `a.x` is **float**, and the type of `a.y` is **int**.

You cannot give default arguments to the same template parameters in different declarations in the same scope. For example, the compiler will not allow the following:

```
template<class T = char> class X;
template<class T = char> class X { };
```

If one template parameter has a default argument, then all template parameters following it must also have default arguments. For example, the compiler will not allow the following:

```
template<class T = char, class U, class V = int> class X { };
```

Template parameter `U` needs a default argument or the default for `T` must be removed.

The scope of a template parameter starts from the point of its declaration to the end of its template definition. This implies that you may use the name of a template parameter in other template parameter declarations and their default arguments. The following example demonstrates this:

```
template<class T = int> class A;
template<class T = float> class B;
template<class V, V obj> class C;
// a template parameter (T) used as the default argument
// to another template parameter (U)
template<class T, class U = T> class D { };
```

Template Arguments

▶ C++ There are three kinds of template arguments corresponding to the three types of template parameters:

- type
- non-type
- template

A template argument must match the type and form specified by the corresponding parameter declared in the template.

To use the default value of a template parameter, you omit the corresponding template argument. However, even if all template parameters have defaults, you still must use the <> brackets. For example, the following will yield a syntax error:

```
template<class T = int> class X { };
X<> a;
X b;
```

The last declaration, `X b`, will yield an error.

Template Type Arguments

▶ C++ You cannot use one of the following as a template argument for a type template parameter:

- a local type
- a type with no linkage
- an unnamed type
- a type compounded from any of the above types

If it is ambiguous whether a template argument is a type or an expression, the template argument is considered to be a type. The following example demonstrates this:

```
template<class T> void f() { };
template<int i> void f() { };

int main() {
  f<int()>();
}
```

The function call `f<int()>()` calls the function with `T` as a template argument – the compiler considers `int()` as a type – and therefore implicitly instantiates and calls the first `f()`.

Template Non-Type Arguments

▶ C++ A non-type template argument provided within a template argument list is an expression whose value can be determined at compile time. Such arguments must be constant expressions, addresses of functions or objects with external linkage, or addresses of static class members. Non-type template arguments are normally used to initialize a class or to specify the sizes of class members.

For non-type integral arguments, the instance argument matches the corresponding template parameter as long as the instance argument has a value and sign appropriate to the parameter type.

For non-type address arguments, the type of the instance argument must be of the form `identifier` or `&identifier`, and the type of the instance argument must match the template parameter exactly, except that a function name is changed to a pointer to function type before matching.

The resulting values of non-type template arguments within a template argument list form part of the template class type. If two template class names have the same template name and if their arguments have identical values, they are the same class.

In the following example, a class template is defined that requires a non-type template **int** argument as well as the type argument:

```
template<class T, int size> class myfilebuf
{
      T* filepos;
      static int array[size];
public:
      myfilebuf() { /* ... */ }
      ~myfilebuf();
      advance(); // function defined elsewhere in program
};
```

In this example, the template argument `size` becomes a part of the template class name. An object of such a template class is created with both the type argument `T` of the class and the value of the non-type template argument `size`.

An object `x`, and its corresponding template class with arguments **double** and `size=200`, can be created from this template with a value as its second template argument:

```
myfilebuf<double,200> x;
```

`x` can also be created using an arithmetic expression:

```
myfilebuf<double,10*20> x;
```

The objects created by these expressions are identical because the template arguments evaluate identically. The value `200` in the first expression could have been represented by an expression whose result at compile time is known to be equal to `200`, as shown in the second construction.

Note: Arguments that contain the < symbol or the > symbol must be enclosed in parentheses to prevent either symbol from being parsed as a template argument list delimiter when it is in fact being used as a relational operator. For example, the arguments in the following definition are valid:

```
myfilebuf<double, (75>25)> x;       // valid
```

The following definition, however, is not valid because the greater than operator (>) is interpreted as the closing delimiter of the template argument list:

```
      myfilebuf<double, 75>25> x;         // error
```

If the template arguments do not evaluate identically, the objects created are of different types:

```
myfilebuf<double,200> x;            // create object x of class
                                    // myfilebuf<double,200>
myfilebuf<double,200.0> y;          // error, 200.0 is a double,
                                    // not an int
```

The instantiation of y fails because the value 200.0 is of type **double**, and the template argument is of type **int**.

The following two objects:

```
      myfilebuf<double, 128> x
      myfilebuf<double, 512> y
```

are objects of separate template specializations. Referring either of these objects later with myfilebuf<double> is an error.

A class template does not need to have a type argument if it has non-type arguments. For example, the following template is a valid class template:

```
template<int i> class C
{
      public:
            int k;
            C() { k = i; }
};
```

This class template can be instantiated by declarations such as:

```
class C<100>;
class C<200>;
```

Again, these two declarations refer to distinct classes because the values of their non-type arguments differ.

Template Template Arguments

C++ A template argument for a template template parameter is the name of a class template.

When the compiler tries to find a template to match the template template argument, it only considers primary class templates. (A *primary template* is the template that is being specialized.) The compiler will not consider any partial specialization even if their parameter lists match that of the template template parameter. For example, the compiler will not allow the following code:

```
template<class T, int i> class A {
   int x;
};

template<class T> class A<T, 5> {
   short x;
};

template<template<class T> class U> class B1 { };

B1<A> c;
```

The compiler will not allow the declaration B1<A> c. Although the partial specialization of A seems to match the template template parameter U of B1, the compiler considers only the primary template of A, which has different template parameters than U.

The compiler considers the partial specializations based on a template template argument once you have instantiated a specialization based on the corresponding template template parameter. The following example demonstrates this:

```
#include <iostream>
using namespace std;

template<class T, class U> class A {
   int x;
};

template<class U> class A<int, U> {
   short x;
};

template<template<class T, class U> class V> class B {
   V<int, char> i;
   V<char, char> j;
};

B<A> c;

int main() {
   cout << typeid(c.i.x).name() << endl;
   cout << typeid(c.j.x).name() << endl;
}
```

The following is the output of the above example:

```
short
int
```

The declaration V<int, char> i uses the partial specialization while the declaration V<char, char> j uses the primary template.

Class Templates

C++ The relationship between a class template and an individual class is like the relationship between a class and an individual object. An individual class defines how a group of objects can be constructed, while a class template defines how a group of classes can be generated.

Note the distinction between the terms *class template* and *template class*:

Class template is a template used to generate template classes. You cannot declare an object of a class template.

Template class is an instance of a class template.

A template definition is identical to any valid class definition that the template might generate, except for the following:

- The class template definition is preceded by

  ```
  template< template-parameter-list >
  ```

  where *template-parameter-list* is a comma-separated list of one or more of the following kinds of template parameters:
  – type

  - non-type
  - template
- Types, variables, constants and objects within the class template can be declared using the template parameters as well as explicit types (for example, **int** or **char**).

A class template can be declared without being defined by using an elaborated type specifier. For example:

```
template<class L,class T> class key;
```

This reserves the name as a class template name. All template declarations for a class template must have the same types and number of template arguments. Only one template declaration containing the class definition is allowed.

Note: When you have nested template argument lists, you must have a separating space between the > at the end of the inner list and the > at the end of the outer list. Otherwise, there is an ambiguity between the output operator >> and two template list delimiters >.

```
template<class L,class T> class key { /* ... */};
template<class L> class vector { /* ... */ };

int main ()
{
   class key <int, vector<int> > my_key_vector;
   // implicitly instantiates template
}
```

Objects and function members of individual template classes can be accessed by any of the techniques used to access ordinary class member objects and functions. Given a class template:

```
template<class T> class vehicle
{
public:
    vehicle() { /* ... */ }    // constructor
    ~vehicle() {};             // destructor
    T kind[16];
    T* drive();
    static void roadmap();
    // ...
};
```

and the declaration:

```
vehicle<char> bicycle; // instantiates the template
```

the constructor, the constructed object, and the member function `drive()` can be accessed with any of the following (assuming the standard header file `<string.h>` is included in the program file):

constructor	`vehicle<char> bicycle;`
	
	`// constructor called automatically,`
	`// object bicycle created`

object `bicycle`	`strcpy (bicycle.kind, "10 speed");`
	`bicycle.kind[0] = '2';`

function `drive()`	`char* n = bicycle.drive();`
function `roadmap()`	`vehicle<char>::roadmap();`

Class Template Declarations and Definitions

C++ A class template must be declared before any declaration of a corresponding template class. A class template definition can only appear once in any single translation unit. A class template must be defined before any use of a template class that requires the size of the class or refers to members of the class.

In the following example, the class template `key` is declared before it is defined. The declaration of the pointer `keyiptr` is valid because the size of the class is not needed. The declaration of `keyi`, however, causes an error.

```
template  <class L> class key;      // class template declared,
                                    // not defined yet
                                    //
class key<int> *keyiptr;            // declaration of pointer
                                    //
class key<int> keyi;                // error, cannot declare keyi
                                    // without knowing size
                                    //
template <class L> class key        // now class template defined
{ /* ... */ };
```

If a template class is used before the corresponding class template is defined, the compiler issues an error. A class name with the appearance of a template class name is considered to be a template class. In other words, angle brackets are valid in a class name only if that class is a template class.

The previous example uses the elaborated type specifier **class** to declare the class template `key` and the pointer `keyiptr`. The declaration of `keyiptr` can also be made without the elaborated type specifier.

```
template  <class L> class key;            // class template declared,
                                          // not defined yet
                                          //
key<int> *keyiptr;                        // declaration of pointer
                                          //
key<int> keyi;                            // error, cannot declare keyi
                                          // without knowing size
                                          //
template <class L> class key              // now class template defined
{ /* ... */ };
```

Static Data Members and Templates

C++ Each class template instantiation has its own copy of any static data members. The static declaration can be of template argument type or of any defined type.

You must separately define static members. The following example demonstrates this:

```
template <class T> class K
{
public:
      static T x;
};
template <class T> T K<T> ::x;

int main()
{
      K<int>::x = 0;
}
```

The statement `template T K::x` defines the static member of class `K`, while the statement in the `main()` function assigns a value to the data member for `K <int>`.

Member Functions of Class Templates

C++ You may define a template member function outside of its class template definition.

When you call a member function of a class template specialization, the compiler will use the template arguments that you used to generate the class template. The following example demonstrates this:

```
template<class T> class X {
   public:
      T operator+(T);
};

template<class T> T X<T>::operator+(T arg1) {
   return arg1;
};

int main() {
   X<char> a;
   X<int> b;
   a +'z';
   b + 4;
}
```

The overloaded addition operator has been defined outside of class `X`. The statement `a + 'z'` is equivalent to `a.operator+('z')`. The statement `b + 4` is equivalent to `b.operator+(4)`.

Friends and Templates

C++ There are four kinds of relationships between classes and their friends when templates are involved:

- *One-to-many*: A non-template function may be a friend to all template class instantiations.
- *Many-to-one*: All instantiations of a template function may be friends to a regular non-template class.
- *One-to-one*: A template function instantiated with one set of template arguments may be a friend to one template class instantiated with the same set of template arguments. This is also the relationship between a regular non-template class and a regular non-template friend function.
- *Many-to-many*: All instantiations of a template function may be a friend to all instantiations of the template class.

The following example demonstrates these relationships:

```
class B{
   template<class V> friend int j();
}

template<class S> g();

template<class T> class A {
   friend int e();
   friend int f(T);
   friend int g<T>();
   template<class U> friend int h();
};
```

- Function `e()` has a one-to-many relationship with class `A`. Function `e()` is a friend to all instantiations of class `A`.

- Function f() has a one-to-one relationship with class A. The compiler will give you a warning for this kind of declaration similar to the following:

```
The friend function declaration "f" will cause an error when the enclosing
template class is instantiated with arguments that declare a friend function
that does not match an existing definition. The function declares only one
function because it is not a template but the function type depends on
one or more template parameters.
```

- Function g() has a one-to-one relationship with class A. Function g() is a function template. It must be declared before here or else the compiler will not recognize g<T> as a template name. For each instantiation of A there is one matching instantiation of g(). For example, g<int> is a friend of A<int>.
- Function h() has a many-to-many relationship with class A. Function h() is a function template. For all instantiations of A all instantiations of h() are friends.
- Function j() has a many-to-one relationship with class B.

These relationships also apply to friend classes.

Function Templates

C++ A *function template* defines how a group of functions can be generated.

A non-template function is not related to a function template, even though the non-template function may have the same name and parameter profile as those of a specialization generated from a template. A non-template function is never considered to be a specialization of a function template.

The following example implements the QuickSort algorithm with a function template named quicksort:

```
#include <iostream>
#include <cstdlib>
using namespace std;

template<class T> void quicksort(T a[], const int& leftarg, const int& rightarg)
{
  if (leftarg < rightarg) {

    T pivotvalue = a[leftarg];
    int left = leftarg - 1;
    int right = rightarg + 1;

  for(;;) {

    while (a[--right] > pivotvalue);
    while (a[++left] < pivotvalue);

    if (left >= right) break;

    T temp = a[right];
    a[right] = a[left];
    a[left] = temp;
  }

  int pivot = right;
  quicksort(a, leftarg, pivot);
  quicksort(a, pivot + 1, rightarg);
  }
}

int main(void) {
  int sortme[10];

  for (int i = 0; i < 10; i++) {
    sortme[i] = rand();
    cout << sortme[i] << " ";
  };
  cout << endl;

  quicksort<int>(sortme, 0, 10 - 1);

  for (int i = 0; i < 10; i++) cout << sortme[i] << "
  ";
  cout << endl;
  return 0;
}
```

The above example will have output similar to the following:

```
16838 5758 10113 17515 31051 5627 23010 7419 16212 4086
4086 5627 5758 7419 10113 16212 16838 17515 23010 31051
```

This QuickSort algorithm will sort an array of type `T` (whose relational and assignment operators have been defined). The template function takes one template argument and three function arguments:

- the type of the array to be sorted, `T`
- the name of the array to be sorted, `a`
- the lower bound of the array, `leftarg`
- the upper bound of the array, `rightarg`

In the above example, you can also call the `quicksort()` template function with the following statement:

```
  quicksort(sortme, 0, 10 - 1);
```

You may omit any template argument if the compiler can deduce it by the usage and context of the template function call. In this case, the compiler deduces that `sortme` is an array of type **int**.

Template Argument Deduction

▶ C++ When you call a template function, you may omit any template argument that the compiler can determine or *deduce* by the usage and context of that template function call.

The compiler tries to deduce a template argument by comparing the type of the corresponding template parameter with the type of the argument used in the function call. The two types that the compiler compares (the template parameter and the argument used in the function call) must be of a certain structure in order for template argument deduction to work. The following lists these type structures:

```
T
const T
volatile T
T&
T*
T[10]
A<T>
C(*)(T)
T(*)()
T(*)(U)
T C::*
C T::*
T U::*
T (C::*)()
C (T::*)()
D (C::*)(T)
```

```
C (T::*)(U)
T (C::*)(U)
T (U::*)()
T (U::*)(V)
E[10][i]
B<i>
TT<T>
TT<i>
TT<C>
```

- T, U, and V represent a template type argument
- 10 represents any integer constant
- i represents a template non-type argument
- [i] represents an array bound of a reference or pointer type, or a non-major array bound of a normal array.
- TT represents a template template argument
- (T), (U), and (V) represents an argument list that has at least one template type argument
- () represents an argument list that has no template arguments
- <T> represents a template argument list that has at least one template type argument
- <i> represents a template argument list that has at least one template non-type argument
- <C> represents a template argument list that has no template arguments dependent on a template parameter

The following example demonstrates the use of each of these type structures. The example declares a template function using each of the above structures as an argument. These functions are then called (without template arguments) in order of declaration. The example outputs the same list of type structures:

```
#include <iostream>
using namespace std;

template<class T> class A { };
template<int i> class B { };

class C {
   public:
      int x;
};

class D {
   public:
      C y;
      int z;
};

template<class T> void f (T)          { cout << "T" << endl; };
template<class T> void f1(const T)    { cout << "const T" << endl; };
template<class T> void f2(volatile T) { cout << "volatile T" << endl;  };
template<class T> void g (T*)         { cout << "T*" << endl; };
template<class T> void g (T&)         { cout << "T&" << endl; };
template<class T> void g1(T[10])      { cout << "T[10]" << endl;};
template<class T> void h1(A<T>)       { cout << "A<T>" << endl; };

void test_1() {
   A<char> a;
   C c;

   f(c);   f1(c);   f2(c);
   g(c);   g(&c);   g1(&c);
   h1(a);
}
```

```
template<class T>          void j(C(*)(T)) { cout << "C(*) (T)" << endl; };
template<class T>          void j(T(*)())  { cout << "T(*) ()" << endl; }
template<class T, class U> void j(T(*)(U)) { cout << "T(*) (U)" << endl; };

void test_2() {
   C (*c_pfunct1)(int);
   C (*c_pfunct2)(void);
   int (*c_pfunct3)(int);
   j(c_pfunct1);
   j(c_pfunct2);
   j(c_pfunct3);
}

template<class T>          void k(T C::*) { cout << "T C::*" << endl; };
template<class T>          void k(C T::*) { cout << "C T::*" << endl; };
template<class T, class U> void k(T U::*) { cout << "T U::*" << endl; };

void test_3() {
   k(&C::x);
   k(&D::y);
   k(&D::z);
}

template<class T>     void m(T (C::*)() )
   { cout << "T (C::*)()" << endl; };
template<class T>     void m(C (T::*)() )
   { cout << "C (T::*)()" << endl; };
template<class T>     void m(D (C::*)(T))
   { cout << "D (C::*)(T)" << endl; };
template<class T, class U>  void m(C (T::*)(U))
   { cout << "C (T::*)(U)" << endl; };
template<class T, class U>  void m(T (C::*)(U))
   { cout << "T (C::*)(U)" << endl; };
template<class T, class U>  void m(T (U::*)() )
   { cout << "T (U::*)()" << endl; };
template<class T, class U, class V> void m(T (U::*)(V))
   { cout << "T (U::*)(V)" << endl; };

void test_4() {
   int (C::*f_membp1)(void);
   C (D::*f_membp2)(void);
   D (C::*f_membp3)(int);
   m(f_membp1);
   m(f_membp2);
   m(f_membp3);

   C (D::*f_membp4)(int);
   int (C::*f_membp5)(int);
   int (D::*f_membp6)(void);
   m(f_membp4);
   m(f_membp5);
   m(f_membp6);

   int (D::*f_membp7)(int);
   m(f_membp7);
}

template<int i> void n(C[10][i]) { cout << "E[10][i]" << endl; };
template<int i> void n(B<i>)     { cout << "B<i>" << endl; };

void test_5() {
   C array[10][20];
   n(array);
   B<20> b;
   n(b);
}
```

```
template<template<class> class TT, class T> void p1(TT<T>)
   { cout << "TT<T>" << endl; };
template<template<int> class TT, int i>     void p2(TT<i>)
   { cout << "TT<i>" << endl; };
template<template<class> class TT>          void p3(TT<C>)
   { cout << "TT<C>" << endl; };

void test_6() {
   A<char> a;
   B<20> b;
   A<C> c;
   p1(a);
   p2(b);
   p3(c);
}

int main() { test_1(); test_2(); test_3(); test_4(); test_5(); test_6(); }
```

Deducing Type Template Arguments

C++ The compiler can deduce template arguments from a type composed of several of the listed type structures. The following example demonstrates template argument deduction for a type composed of several type structures:

```
template<class T> class Y { };

template<class T, int i> class X {
   public:
      Y<T> f(char[20][i]) { return x; };
      Y<T> x;
};

template<template<class> class T, class U, class V, class W, int i>
   void g( T<U> (V::*)(W[20][i]) ) { };

int main()
{
   Y<int> (X<int, 20>::*p)(char[20][20]) = &X<int, 20>::f;
   g(p);
}
```

The type `Y<int> (X<int, 20>::*p)(char[20][20])T<U> (V::*)(W[20][i])` is based on the type structure `T (U::*)(V)`:

- `T` is `Y<int>`
- `U` is `X<int, 20>`
- `V` is `char[20][20]`

If you qualify a type with the class to which that type belongs, and that class (a *nested name specifier*) depends on a template parameter, the compiler will not deduce a template argument for that parameter. If a type contains a template argument that cannot be deduced for this reason, all template arguments in that type will not be deduced. The following example demonstrates this:

```
template<class T, class U, class V>
  void h(typename Y<T>::template Z<U>, Y<T>, Y<V>) { };

int main() {
  Y<int>::Z<char> a;
  Y<int> b;
  Y<float> c;

  h<int, char, float>(a, b, c);
  h<int, char>(a, b, c);
  // h<int>(a, b, c);
}
```

The compiler will not deduce the template arguments T and U in typename Y<T>::template Z<U> (but it will deduce the T in Y<T>). The compiler would not allow the template function call h<int>(a, b, c) because U is not deduced by the compiler.

The compiler can deduce a function template argument from a pointer to function or pointer to member function argument given several overloaded function names. However, none of the overloaded functions may be function templates, nor can more than one overloaded function match the required type. The following example demonstrates this:

```
template<class T> void f(void(*) (T,int)) { };

template<class T> void g1(T, int) { };

void g2(int, int) { };
void g2(char, int) { };

void g3(int, int, int) { };
void g3(float, int) { };

int main() {
//   f(&g1);
//   f(&g2);
   f(&g3);
}
```

The compiler would not allow the call f(&g1) because g1() is a function template. The compiler would not allow the call f(&g2) because both functions named g2() match the type required by f().

The compiler cannot deduce a template argument from the type of a default argument. The following example demonstrates this:

```
template<class T> void f(T = 2, T = 3) { };

int main() {
   f(6);
//   f();
   f<int>();
}
```

The compiler allows the call f(6) because the compiler deduces the template argument (**int**) by the value of the function call's argument. The compiler would not allow the call f() because the compiler cannot deduce template argument from the default arguments of f().

The compiler cannot deduce a template type argument from the type of a non-type template argument. For example, the compiler will not allow the following:

```
template<class T, T i> void f(int[20][i]) { };

int main() {
   int a[20][30];
   f(a);
}
```

The compiler cannot deduce the type of template parameter T.

Deducing Non-Type Template Arguments

C++ The compiler cannot deduce the value of a major array bound unless the bound refers to a reference or pointer type. Major array bounds are not part of function parameter types. The following code demonstrates this:

```
template<int i> void f(int a[10][i]) { };
template<int i> void g(int a[i]) { };
template<int i> void h(int (&a)[i]) { };

int main () {
   int b[10][20];
   int c[10];
   f(b);
   // g(c);
   h(c);
}
```

The compiler would not allow the call `g(c)`; the compiler cannot deduce template argument `i`.

The compiler cannot deduce the value of a non-type template argument used in an expression in the template function's parameter list. The following example demonstrates this:

```
template<int i> class X { };

template<int i> void f(X<i - 1>) { };

int main () {
  X<0> a;
  f<1>(a);
  // f(a);
}
```

In order to call function `f()` with object `a`, the function must accept an argument of type `X<0>`. However, the compiler cannot deduce that the template argument `i` must be equal to 1 in order for the function template argument type `X<i - 1>` to be equivalent to `X<0>`. Therefore the compiler would not allow the function call `f(a)`.

If you want the compiler to deduce a non-type template argument, the type of the parameter must match exactly the type of value used in the function call. For example, the compiler will not allow the following:

```
template<int i> class A { };
template<short d> void f(A<d>) { };

int main() {
   A<1> a;
   f(a);
}
```

The compiler will not convert **int** to **short** when the example calls `f()`.

However, deduced array bounds may be of any integral type.

Overloading Function Templates

C++ You may overload a function template either by a non-template function or by another function template.

If you call the name of an overloaded function template, the compiler will try to deduce its template arguments and check its explicitly declared template arguments. If successful, it will instantiate a function template specialization, then add this specialization to the set of *candidate functions* used in overload resolution. The compiler proceeds with overload resolution, choosing the most appropriate function from the set of candidate functions. Non-template functions take precedence over template functions. The following example describes this:

```
#include <iostream>
using namespace std;

template<class T> void f(T x, T y) { cout << "Template" << endl; }

void f(int w, int z) { cout << "Non-template" << endl; }

int main() {
   f( 1 ,  2 );
   f('a', 'b');
   f( 1 , 'b');
}
```

The following is the output of the above example:

```
Non-template
Template
Non-template
```

The function call `f(1, 2)` could match the argument types of both the template function and the non-template function. The non-template function is called because a non-template function takes precedence in overload resolution.

The function call `f('a', 'b')` can only match the argument types of the template function. The template function is called.

Argument deduction fails for the function call `f(1, 'b')`; the compiler does not generate any template function specialization and overload resolution does not take place. The non-template function resolves this function call after using the standard conversion from **char** to **int** for the function argument `'b'`.

Partial Ordering of Function Templates

C++ A function template specialization might be ambiguous because template argument deduction might associate the specialization with more than one of the overloaded definitions. The compiler will then choose the definition that is the most specialized. This process of selecting a function template definition is called *partial ordering*.

A template `X` is more specialized than a template `Y` if every argument list that matches the one specified by `X` also matches the one specified by `Y`, but not the other way around. The following example demonstrates partial ordering:

```
template<class T> void f(T)  { }
template<class T> void f(T*) { }
template<class T> void f(const T*) { }

template<class T> void g(T) { }
template<class T> void g(T&) { }

template<class T> void h(T) { }
template<class T> void h(T, ...) { }

int main() {
   const int *p;
   f(p);
   int q;
//  g(q);
//  h(q);
}
```

The declaration `template<class T> void f(const T*)` is more specialized than `template<class T> void f(T*)`. Therefore, the function call `f(p)` calls `template<class T> void f(const T*)`. However, neither `void g(T)` nor `void g(T&)` is more specialized than the other. Therefore, the function call `g(q)` would be ambiguous.

Ellipses do not affect partial ordering. Therefore, the function call `h(q)` would also be ambiguous.

The compiler uses partial ordering in the following cases:
- Calling a function template specialization that requires overload resolution.
- Taking the address of a function template specialization.
- When a friend function declaration, an explicit instantiation, or explicit specialization refers to a function template specialization.
- Determining the appropriate deallocation function that is also a function template for a given placement operator new.

Template Instantiation

C++ The act of creating a new definition of a function, class, or member of a class from a template declaration and one or more template arguments is called *template instantiation*. The definition created from a template instantiation is called a *specialization*.

A forward declaration of a template instantiation has the form of an explicit template instantiation preceded by the **extern** keyword.

```
►►──extern──template──template_declaration─────────────────────────────►◄
```

The language feature is an orthogonal extension to Standard C++ and C++98 for compatibility with GNU C++.

Implicit Instantiation

C++ Unless a template specialization has been explicitly instantiated or explicitly specialized, the compiler will generate a specialization for the template only when it needs the definition. This is called *implicit instantiation*.

If the compiler must instantiate a class template specialization and the template is declared, you must also define the template.

For example, if you declare a pointer to a class, the definition of that class is not needed and the class will not be implicitly instantiated. The following example demonstrates when the compiler instantiates a template class:

```
template<class T> class X {
  public:
    X* p;
    void f();
    void g();
};

X<int>* q;
```

```
X<int> r;
X<float>* s;
r.f();
s->g();
```

The compiler requires the instantiation of the following classes and functions:
- X<int> when the object r is declared
- X<int>::f() at the member function call r.f()
- X<float> and X<float>::g() at the class member access function call s->g()

Therefore, the functions X<T>::f() and X<T>::g() must be defined in order for the above example to compile. (The compiler will use the default constructor of class X when it creates object r.) The compiler does not require the instantiation of the following definitions:
- class X when the pointer p is declared
- X<int> when the pointer q is declared
- X<float> when the pointer s is declared

The compiler will implicitly instantiate a class template specialization if it is involved in pointer conversion or pointer to member conversion. The following example demonstrates this:

```
template<class T> class B { };
template<class T> class D : public B<T> { };

void g(D<double>* p, D<int>* q)
{
  B<double>* r = p;
  delete q;
}
```

The assignment B<double>* r = p converts p of type D<double>* to a type of B<double>*; the compiler must instantiate D<double>. The compiler must instantiate D<int> when it tries to delete q.

If the compiler implicitly instantiates a class template that contains static members, those static members are not implicitly instantiated. The compiler will instantiate a static member only when the compiler needs the static member's definition. Every instantiated class template specialization has its own copy of static members. The following example demonstrates this:

```
template<class T> class X {
public:
   static T v;
};

template<class T> T X<T>::v = 0;

X<char*> a;
X<float> b;
X<float> c;
```

Object a has a static member variable v of type char*. Object b has a static variable v of type float. Objects b and c share the single static data member v.

An implicitly instantiated template is in the same namespace where you defined the template.

If a function template or a member function template specialization is involved with overload resolution, the compiler implicitly instantiates a declaration of the specialization.

Explicit Instantiation

C++ You can explicitly tell the compiler when it should generate a definition from a template. This is called *explicit instantiation*.

Syntax – Explicit Instantiation Declaration

```
►►──template──template_declaration─────────────────────────────────────►◄
```

The following are examples of explicit instantiations:

```
template<class T> class Array { void mf(); };
template class Array<char>;      // explicit instantiation
template void Array<int>::mf();  // explicit instantiation

template<class T> void sort(Array<T>& v) { }
template void sort(Array<char>&); // explicit instantiation

namespace N {
   template<class T> void f(T&) { }
}

template void N::f<int>(int&);
// The explicit instantiation is in namespace N.

int* p = 0;
template<class T> T g(T = &p);
template char g(char);                 // explicit instantiation

template <class T> class X {
   private:
      T v(T arg) { return arg; };
};

template int X<int>::v(int);    // explicit instantiation

template<class T> T g(T val) { return val;}
template<class T> void Array<T>::mf() { }
```

A template declaration must be in scope at the point of instantiation of the template's explicit instantiation. An explicit instantiation of a template specialization is in the same namespace where you defined the template.

Access checking rules do not apply to names in explicit instantiations. Template arguments and names in a declaration of an explicit instantiation may be private types or objects. In the above example, the compiler allows the explicit instantiation `template int X<int>::v(int)` even though the member function is declared private.

The compiler does not use default arguments when you explicitly instantiate a template. In the above example the compiler allows the explicit instantiation `template char g(char)` even though the default argument is an address of type **int**.

An **extern**-qualified template declaration does not instantiate the class or function. For both classes and functions, the **extern** template instantiation prevents the instantiation of parts of the template, provided that the instantiation has not already been triggered by code prior to the **extern** template instantiation. For classes, the members (both static and nonstatic) are not instantiated. The class itself is instantiated if required to map the class. For functions, the prototype is instantiated, but the body of the template function is not instantiated.

The following examples show template instantiation using **extern**:

```
template<class T>class C {
   static int i;
   void f(T) { }
};
template<class U>int C<U>::i = 0;
extern template C<int>;  // extern explicit template instantiation
C<int>c;   // does not cause instantiation of C<int>::i
           // or C<int>::f(int) in this file,
           // but the class is instantiated for mapping
C<char>d;  // normal instantiations
```

```
template<class C> C foo(C c) { return c; }
extern template int foo<int>(int);  // extern explicit template instantiation
int i = foo(1);  // does not cause instantiation of the body of foo<int>
```

Template Specialization

C++ The act of creating a new definition of a function, class, or member of a class from a template declaration and one or more template arguments is called *template instantiation*. The definition created from a template instantiation is called a *specialization*. A *primary template* is the template that is being specialized.

Explicit Specialization

C++ When you instantiate a template with a given set of template arguments the compiler generates a new definition based on those template arguments. You can override this behavior of definition generation. You can instead specify the definition the compiler uses for a given set of template arguments. This is called *explicit specialization*. You can explicitly specialize any of the following:

- Function or class template
- Member function of a class template
- Static data member of a class template
- Member class of a class template
- Member function template of a class template
- Member class template of a class template

Syntax – Explicit

```
►►──template──<──>──declaration_name──┬──────────────────────────────┬──declaration_body──────►◄
                                      └─<──template_argument_list──>─┘
```

The **template<>** prefix indicates that the following template declaration takes no template parameters. The *declaration_name* is the name of a previously declared template. Note that you can forward-declare an explicit specialization so the *declaration_body* is optional, at least until the specialization is referenced.

The following example demonstrates explicit specialization:

```
using namespace std;

template<class T = float, int i = 5> class A
{
   public:
      A();
      int value;
};

template<> class A<> { public: A(); };
template<> class A<double, 10> { public: A(); };
```

```
template<class T, int i> A<T, i>::A() : value(i) {
   cout << "Primary template, "
        << "non-type argument is " << value << endl;
}

A<>::A() {
   cout << "Explicit specialization "
        << "default arguments" << endl;
}

A<double, 10>::A() {
   cout << "Explicit specialization "
        << "<double, 10>" << endl;
}

int main() {
   A<int,6> x;
   A<> y;
   A<double, 10> z;
}
```

The following is the output of the above example:

```
Primary template non-type argument is: 6
Explicit specialization default arguments
Explicit specialization <double, 10>
```

This example declared two explicit specializations for the *primary template* (the template which is being specialized) class `A`. Object `x` uses the constructor of the primary template. Object `y` uses the explicit specialization `A<>::A()`. Object `z` uses the explicit specialization `A<double, 10>::A()`.

Definition and Declaration of Explicit Specializations

C++ The definition of an explicitly specialized class is unrelated to the definition of the primary template. You do not have to define the primary template in order to define the specialization (nor do you have to define the specialization in order to define the primary template). For example, the compiler will allow the following:

```
template<class T> class A;
template<> class A<int>;

template<> class A<int> { /* ... */ };
```

The primary template is not defined, but the explicit specialization is.

You can use the name of an explicit specialization that has been declared but not defined the same way as an incompletely defined class. The following example demonstrates this:

```
template<class T> class X { };
template<>  class X<char>;
X<char>* p;
X<int> i;
// X<char> j;
```

The compiler does not allow the declaration `X<char> j` because the explicit specialization of `X<char>` is not defined.

Explicit Specialization and Scope

C++ A declaration of a primary template must be in scope at the *point of declaration* of the explicit specialization. In other words, an explicit specialization declaration must appear after the declaration of the primary template. For example, the compiler will not allow the following:

```
template<> class A<int>;
template<class T> class A;
```

An explicit specialization is in the same namespace as the definition of the primary template.

Class Members of Explicit Specializations

C++ A member of an explicitly specialized class is not implicitly instantiated from the member declaration of the primary template. You have to explicitly define members of a class template specialization. You define members of an explicitly specialized template class as you would normal classes, without the **template<>** prefix. In addition, you can define the members of an explicit specialization inline; no special template syntax is used in this case. The following example demonstrates a class template specialization:

```
template<class T> class A {
   public:
      void f(T);
};

template<> class A<int> {
   public:
      int g(int);
};

int A<int>::g(int arg) { return 0; }

int main() {
   A<int> a;
   a.g(1234);
}
```

The explicit specialization `A<int>` contains the member function `g()`, which the primary template does not.

If you explicitly specialize a template, a member template, or the member of a class template, then you must declare this specialization before that specialization is implicitly instantiated. For example, the compiler will not allow the following code:

```
template<class T> class A { };

void f() { A<int> x; }
template<> class A<int> { };

int main() { f(); }
```

The compiler will not allow the explicit specialization `template<> class A<int> { };` because function `f()` uses this specialization (in the construction of `x`) before the specialization.

Explicit Specialization of Function Templates

C++ In a function template specialization, a template argument is optional if the compiler can deduce it from the type of the function arguments. The following example demonstrates this:

```
template<class T> class X { };
template<class T> void f(X<T>);
template<> void f(X<int>);
```

The explicit specialization `template<> void f(X<int>)` is equivalent to `template<> void f<int>(X<int>)`.

You cannot specify default function arguments in a declaration or a definition for any of the following:

- Explicit specialization of a function template
- Explicit specialization of a member function template

For example, the compiler will not allow the following code:

```
template<class T> void f(T a) { };
template<> void f<int>(int a = 5) { };

template<class T> class X {
  void f(T a) { }
};
template<> void X<int>::f(int a = 10) { };
```

Explicit Specialization of Members of Class Templates

▶ C++ Each instantiated class template specialization has its own copy of any static members. You may explicitly specialize static members. The following example demonstrates this:

```
template<class T> class X {
public:
   static T v;
   static void f(T);
};

template<class T> T X<T>::v = 0;
template<class T> void X<T>::f(T arg) { v = arg; }

template<> char* X<char*>::v = "Hello";
template<> void X<float>::f(float arg) { v = arg * 2; }

int main() {
   X<char*> a, b;
   X<float> c;
   c.f(10);
}
```

This code explicitly specializes the initialization of static data member `X::v` to point to the string `"Hello"` for the template argument **char***. The function `X::f()` is explicitly specialized for the template argument **float**. The static data member `v` in objects `a` and `b` point to the same string, `"Hello"`. The value of `c.v` is equal to 20 after the call function call `c.f(10)`.

You can nest member templates within many enclosing class templates. If you explicitly specialize a template nested within several enclosing class templates, you must prefix the declaration with **template<>** for every enclosing class template you specialize. You may leave some enclosing class templates unspecialized, however you cannot explicitly specialize a class template unless its enclosing class templates are also explicitly specialized. The following example demonstrates explicit specialization of nested member templates:

```
#include <iostream>
using namespace std;

template<class T> class X {
public:
  template<class U> class Y {
  public:
    template<class V> void f(U,V);
    void g(U);
  };
};

template<class T> template<class U> template<class V>
  void X<T>::Y<U>::f(U, V) { cout << "Template 1" <<   endl; }

template<class T> template<class U>
  void X<T>::Y<U>::g(U) { cout << "Template 2" <<   endl; }

template<> template<>
  void X<int>::Y<int>::g(int) { cout << "Template 3"   << endl; }

template<> template<> template<class V>
  void X<int>::Y<int>::f(int, V) { cout << "Template 4" << endl; }

template<> template<> template<>
  void X<int>::Y<int>::f<int>(int, int) { cout << "Template 5" << endl; }

// template<> template<class U> template<class V>
//    void X<char>::Y<U>::f(U, V) { cout << "Template 6" << endl; }

// template<class T> template<>
//    void X<T>::Y<float>::g(float) { cout << "Template 7" << endl; }

int main() {
  X<int>::Y<int> a;
  X<char>::Y<char> b;
  a.f(1, 2);
  a.f(3, 'x');
  a.g(3);
  b.f('x', 'y');
  b.g('z');
}
```

The following is the output of the above program:

```
Template 5
Template 4
Template 3
Template 1
Template 2
```

- The compiler would not allow the template specialization definition that would output "Template 6" because it is attempting to specialize a member (function f()) without specialization its containing class (Y).
- The compiler would not allow the template specialization definition that would output "Template 7" because the enclosing class of class Y (which is class X) is not explicitly specialized.

A friend declaration cannot declare an explicit specialization.

Partial Specialization

C++ When you instantiate a class template, the compiler creates a definition based on the template arguments you have passed. Alternatively, if all those template arguments match those of an explicit specialization, the compiler uses the definition defined by the explicit specialization.

A *partial specialization* is a generalization of explicit specialization. An explicit specialization only has a template argument list. A partial specialization has both a template argument list and a template parameter list. The compiler uses the partial specialization if its template argument list matches a subset of the template arguments of a template instantiation. The compiler will then generate a new definition from the partial specialization with the rest of the unmatched template arguments of the template instantiation.

You cannot partially specialize function templates.

Syntax – Partial Specialization

```
►►──template──<template_parameter_list>──declaration_name──────────────────────►

►─<template_argument_list>──declaration_body──────────────────────────────────►◄
```

The *declaration_name* is a name of a previously declared template. Note that you can forward-declare a partial specialization so that the *declaration_body* is optional.

The following demonstrates the use of partial specializations:

```
#include <iostream>
using namespace std;

template<class T, class U, int I> struct X
  { void f() { cout << "Primary template" << endl; } };

template<class T, int I> struct X<T, T*, I>
  { void f() { cout << "Partial specialization 1" << endl;
  } };

template<class T, class U, int I> struct X<T*, U, I>
  { void f() { cout << "Partial specialization 2" << endl;
  } };

template<class T> struct X<int, T*, 10>
  { void f() { cout << "Partial specialization 3" << endl;
  } };

template<class T, class U, int I> struct X<T, U*, I>
  { void f() { cout << "Partial specialization 4" << endl;
  } };

int main() {
   X<int, int, 10> a;
   X<int, int*, 5> b;
   X<int*, float, 10> c;
   X<int, char*, 10> d;
   X<float, int*, 10> e;
//   X<int, int*, 10> f;
   a.f(); b.f(); c.f(); d.f(); e.f();
}
```

The following is the output of the above example:

```
Primary template
Partial specialization 1
Partial specialization 2
Partial specialization 3
Partial specialization 4
```

The compiler would not allow the declaration `X<int, int*, 10> f` because it can match `template struct X<T, T*, I>`, `template struct X<int, T*, 10>`, or `template struct X<T, U*, I>`, and none of these declarations are a better match than the others.

Each class template partial specialization is a separate template. You must provide definitions for each member of a class template partial specialization.

Template Parameter and Argument Lists of Partial Specializations

C++ Primary templates do not have template argument lists; this list is implied in the template parameter list.

Template parameters specified in a primary template but not used in a partial specialization are omitted from the template parameter list of the partial specialization. The order of a partial specialization's argument list is the same as the order of the primary template's implied argument list.

In a template argument list of a partial template parameter, you cannot have an expression that involves non-type arguments unless that expression is only an identifier. In the following example, the compiler will not allow the first partial specialization, but will allow the second one:

```
template<int I, int J> class X { };

// Invalid partial specialization
template<int I> class X <I * 4, I + 3> { };

// Valid partial specialization
template <int I> class X <I, I> { };
```

The type of a non-type template argument cannot depend on a template parameter of a partial specialization. The compiler will not allow the following partial specialization:

```
template<class T, T i> class X { };

// Invalid partial specialization
template<class T> class X<T, 25> { };
```

A partial specialization's template argument list cannot be the same as the list implied by the primary template.

You cannot have default values in the template parameter list of a partial specialization.

Matching of Class Template Partial Specializations

C++ The compiler determines whether to use the primary template or one of its partial specializations by matching the template arguments of the class template specialization with the template argument lists of the primary template and the partial specializations:

- If the compiler finds only one specialization, then the compiler generates a definition from that specialization.
- If the compiler finds more than one specialization, then the compiler tries to determine which of the specializations is the most specialized. A template `X` is more specialized than a template `Y` if every argument list that matches the one specified by `X` also matches the one specified by `Y`, but not the other way around. If the compiler cannot find the most specialized specialization, then the use of the class template is ambiguous; the compiler will not allow the program.
- If the compiler does not find any matches, then the compiler generates a definition from the primary template.

Name Binding and Dependent Names

C++ *Name binding* is the process of finding the declaration for each name that is explicitly or implicitly used in a template. The compiler may bind a name in the definition of a template, or it may bind a name at the instantiation of a template.

A *dependent name* is a name that depends on the type or the value of a template parameter. For example:

```
template<class T> class U : A<T>
{
  typename T::B x;
  void f(A<T>& y)
  {
    *y++;
  }
};
```

The dependent names in this example are the base class `A<T>`, the type name `T::B`, and the variable `y`.

The compiler binds dependent names when a template is instantiated. The compiler binds non-dependent names when a template is defined. For example:

```
void f(double) { cout << "Function f(double)" << endl; }

template<class T> void g(T a) {
  f(123);
  h(a);
}

void f(int) { cout << "Function f(int)" << endl; }
void h(double) { cout << "Function h(double)" << endl; }

void i() {
  extern void h(int);
  g<int>(234);
}

void h(int) { cout << "Function h(int)" << endl; }
```

The following is the output if you call function `i()`:

```
Function f(double)
Function h(double)
```

The *point of definition* of a template is located immediately before its definition. In this example, the point of definition of the function template `g(T)` is located immediately before the keyword **template**. Because the function call `f(123)` does not depend on a template argument, the compiler will consider names declared before the definition of function template `g(T)`. Therefore, the call `f(123)` will call `f(double)`. Although `f(int)` is a better match, it is not in scope at the point of definition of `g(T)`.

The *point of instantiation* of a template is located immediately before the declaration that encloses its use. In this example, the point of instantiation of the call to `g<int>(234)` is located immediately before `i()`. Because the function call `h(a)` depends on a template argument, the compiler will consider names declared before the instantiation of function template `g(T)`. Therefore, the call `h(a)` will call `h(double)`. It will not consider `h(int)`, because this function was not in scope at the point of instantiation of `g<int>(234)`.

Point of instantiation binding implies the following:

- A template parameter cannot depend on any local name or class member.
- An unqualified name in a template cannot depend on a local name or class member.

The typename Keyword

C++ Use the keyword **typename** if you have a qualified name that refers to a type and depends on a template parameter. Only use the keyword **typename** in template declarations and definitions. The following example illustrates the use of the keyword **typename**:

```
template<class T> class A
{
  T::x(y);
  typedef char C;
  A::C d;
}
```

The statement `T::x(y)` is ambiguous. It could be a call to function `x()` with a nonlocal argument `y`, or it could be a declaration of variable `y` with type `T::x`. C++ will interpret this statement as a function call. In order for the compiler to interpret this statement as a declaration, you would add the keyword **typename** to the beginning of it. The statement `A::C d;` is ill-formed. The class `A` also refers to `A<T>` and thus depends on a template parameter. You must add the keyword **typename** to the beginning of this declaration:

```
  typename A::C d;
```

You can also use the keyword **typename** in place of the keyword **class** in template parameter declarations.

The Keyword template as Qualifier

C++ Use the keyword **template** as a qualifier to distinguish member templates from other names. The following example illustrates when you must use **template** as a qualifier:

```
class A
{
  public:
    template<class T> T function_m() { };
};

template<class U> void function_n(U argument)
{
  char object_x = argument.function_m<char>();
}
```

The declaration `char object_x = argument.function_m<char>();` is ill-formed. The compiler assumes that the < is a less-than operator. In order for the compiler to recognize the function template call, you must add the **template** quantifier:

```
char object_x = argument.template function_m<char>();
```

If the name of a member template specialization appears after a `.`, `->`, or `::` operator, and that name has explicitly qualified template parameters, prefix the member template name with the keyword **template**. The following example demonstrates this use of the keyword **template**:

```
#include <iostream>
using namespace std;

class X {
   public:
      template <int j> struct S {
         void h() {
            cout << "member template's member function: " << j << endl;
         }
      };
      template <int i> void f() {
        cout << "Primary: " << i << endl;
      }
};

template<> void X::f<20>() {
   cout << "Specialized, non-type argument = 20" << endl;
}

template<class T> void g(T* p) {
   p->template f<100>();
   p->template f<20>();
   typename T::template S<40> s; // use of scope operator on a member template
   s.h();
}

int main()
{
   X temp;
   g(&temp);
}
```

The following is the output of the above example:

```
Primary: 100
Specialized, non-type argument = 20
member template's member function: 40
```

If you do not use the keyword **template** in these cases, the compiler will interpret the < as a less-than operator. For example, the following line of code is ill-formed:

```
p->f<100>();
```

The compiler interprets `f` as a non-template member, and the < as a less-than operator.

Chapter 17. Exception Handling

C++ *Exception handling* is a mechanism that separates code that detects and handles exceptional circumstances from the rest of your program. Note that an exceptional circumstance is not necessarily an error.

When a function detects an exceptional situation, you represent this with an object. This object is called an *exception object*. In order to deal with the exceptional situation you *throw the exception*. This passes control, as well as the exception, to a designated block of code in a direct or indirect caller of the function that threw the exception. This block of code is called a *handler*. In a handler, you specify the types of exceptions that it may process. The C++ run time, together with the generated code, will pass control to the first appropriate handler that is able to process the exception thrown. When this happens, an exception is *caught*. A handler may *rethrow* an exception so it can be caught by another handler.

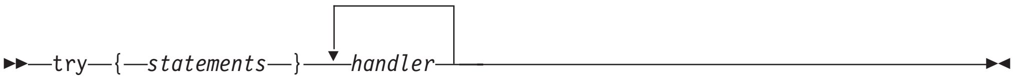
The exception handling mechanism is made up of the following elements:
- **try** blocks: a block of code that may throw an exception that you want to handle with special processing
- **catch** blocks or handlers: a block of code that is executed when a try block encounters an exception
- **throw** expression: indicates when your program encounters an exception
- exception specifications: specify which exceptions (if any) a function may throw
- `unexpected()` function: called when a function throws an exception not specified by an exception specification
- `terminate()` function: called for exceptions that are not caught

Related References
- "The try Keyword"
- "catch Blocks" on page 371
- "The throw Expression" on page 377
- "Exception Specifications" on page 380
- "unexpected()" on page 383
- "terminate()" on page 384

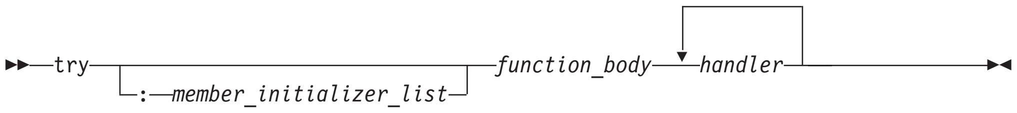
The try Keyword

C++ You use a *try block* to indicate which areas in your program that might throw exceptions you want to handle immediately. You use a *function try block* to indicate that you want to detect exceptions in the entire body of a function.

Syntax – try Block



Syntax — Function try Block



The following is an example of a function try block with a member initializer, a function try block and a try block:

```
#include <iostream>
using namespace std;

class E {
   public:
      const char* error;
      E(const char* arg) : error(arg) { }
};

class A {
   public:
      int i;

      // A function try block with a member
      // initializer
      A() try : i(0) {
         throw E("Exception thrown in A()");
      }
      catch (E& e) {
         cout << e.error << endl;
      }
};

// A function try block
void f() try {
   throw E("Exception thrown in f()");
}
catch (E& e) {
   cout << e.error << endl;
}

void g() {
   throw E("Exception thrown in g()");
}

int main() {
   f();

   // A try block
   try {
      g();
   }
   catch (E& e) {
      cout << e.error << endl;
   }
   try {
      A x;
   }
   catch(...) { }
}
```

The following is the output of the above example:

```
Exception thrown in f()
Exception thrown in g()
Exception thrown in A()
```

The constructor of class `A` has a function try block with a member initializer. Function `f()` has a function try block. The `main()` function contains a try block.

Related References
- “Initializing Base Classes and Members” on page 316

Nested Try Blocks

C++ When try blocks are nested and a **throw** occurs in a function called by an inner try block, control is transferred outward through the nested try blocks until the first catch block is found whose argument matches the argument of the throw expression.

For example:

```
try
{
      func1();
      try
      {
             func2();
      }
      catch (spec_err) { /* ... */ }
      func3();
}
catch (type_err) { /* ... */ }
// if no throw is issued, control resumes here.
```

In the above example, if `spec_err` is thrown within the inner try block (in this case, from `func2()`), the exception is caught by the inner catch block, and, assuming this catch block does not transfer control, `func3()` is called. If `spec_err` is thrown after the inner try block (for instance, by `func3()`), it is not caught and the function `terminate()` is called. If the exception thrown from `func2()` in the inner try block is `type_err`, the program skips out of both try blocks to the second catch block without invoking `func3()`, because no appropriate catch block exists following the inner try block.

You can also nest a try block within a catch block.

Related References
- “terminate()” on page 384
- “unexpected()” on page 383
- “Special Exception Handling Functions” on page 383

catch Blocks

C++ The following is the syntax for an exception handler or a catch block:

```
►►—catch—(—exception_declaration—)—{—statements—}——————————————————►◄
```

You can declare a handler to catch many types of exceptions. The allowable objects that a function can catch are declared in the parentheses following the **catch** keyword (the *exception_declaration*). You can catch objects of the fundamental types, base and derived class objects, references, and pointers to all of these types. You

can also catch **const** and **volatile** types. The *exception_declaration* cannot be an incomplete type, or a reference or pointer to an incomplete type other than one of the following:
- **void***
- **const void***
- **volatile void***
- **const volatile void***

You cannot define a type in an *exception_declaration*.

You can also use the **catch(...)** form of the handler to catch all thrown exceptions that have not been caught by a previous catch block. The ellipsis in the catch argument indicates that any exception thrown can be handled by this handler.

If an exception is caught by a **catch(...)** block, there is no direct way to access the object thrown. Information about an exception caught by **catch(...)** is very limited.

You can declare an optional variable name if you want to access the thrown object in the catch block.

A catch block can only catch accessible objects. The object caught must have an accessible copy constructor.

Related References
- "Type Qualifiers" on page 71
- "Member Access" on page 276

Function try block Handlers

▶ C++ The scope and lifetime of the parameters of a function or constructor extend into the handlers of a function try block. The following example demonstrates this:

```
void f(int &x) try {
   throw 10;
}
catch (const int &i)
{
   x = i;
}

int main() {
   int v = 0;
   f(v);
}
```

The value of `v` after `f()` is called is `10`.

A function try block on `main()` does not catch exceptions thrown in destructors of objects with static storage duration, or constructors of namespace scope objects.

The following example throws an exception from a destructor of a static object:

```
#include <iostream>
using namespace std;

class E {
public:
  const char* error;
  E(const char* arg) : error(arg) { }
};
```

```
class A {
  public: ~A() { throw E("Exception in ~A()"); }
};

class B {
  public: ~B() { throw E("Exception in ~B()"); }
};

int main() try {
  cout << "In main" << endl;
  static A cow;
  B bull;
}
catch (E& e) {
  cout << e.error << endl;
}
```

The following is the output of the above example:

```
In main
Exception in ~B()
```

The run time will not catch the exception thrown when object `cow` is destroyed at the end of the program.

The following example throws an exception from a constructor of a namespace scope object:

```
#include <iostream>
using namespace std;

class E {
public:
  const char* error;
  E(const char* arg) : error(arg) { }
};

namespace N {
  class C {
  public:
    C() {
      cout << "In C()" << endl;
      throw E("Exception in C()");
    }
  };

  C calf;
};

int main() try {
  cout << "In main" << endl;
}
catch (E& e) {
  cout << e.error << endl;
}
```

The following is the output of the above example:

```
In C()
```

The compiler will not catch the exception thrown when object `calf` is created.

In a function try block's handler, you cannot have a jump into the body of a constructor or destructor.

A return statement cannot appear in a function try block's handler of a constructor.

When the function try block's handler of an object's constructor or destructor is entered, fully constructed base classes and members of that object are destroyed. The following example demonstrates this:

```
#include <iostream>
using namespace std;

class E {
   public:
      const char* error;
      E(const char* arg) : error(arg) { };
};

class B {
   public:
      B() { };
      ~B() { cout << "~B() called" << endl; };
};

class D : public B {
   public:
      D();
      ~D() { cout << "~D() called" << endl; };
};

D::D() try : B() {
   throw E("Exception in D()");
}
catch(E& e) {
   cout << "Handler of function try block of D(): " << e.error << endl;
};

int main() {
   try {
      D val;
   }
   catch(...) { }
}
```

The following is the output of the above example:

```
~B() called
Handler of function try block of D(): Exception in D()
```

When the function try block's handler of `D()` is entered, the run time first calls the destructor of the base class of `D`, which is `B`. The destructor of `D` is not called because `val` is not fully constructed.

The run time will rethrow an exception at the end of a function try block's handler of a constructor or destructor. All other functions will return once they have reached the end of their function try block's handler. The following example demonstrates this:

```
#include <iostream>
using namespace std;

class E {
   public:
      const char* error;
      E(const char* arg) : error(arg) { };
};

class A {
   public:
      A() try { throw E("Exception in A()"); }
      catch(E& e) { cout << "Handler in A(): " << e.error << endl; }
};

int f() try {
   throw E("Exception in f()");
   return 0;
}
catch(E& e) {
   cout << "Handler in f(): " << e.error << endl;
   return 1;
}

int main() {
   int i = 0;
   try { A cow; }
   catch(E& e) {
      cout << "Handler in main(): " << e.error << endl;
   }

   try { i = f(); }
   catch(E& e) {
      cout << "Another handler in main(): " << e.error << endl;
   }

   cout << "Returned value of f(): " << i << endl;
}
```

The following is the output of the above example:

```
Handler in A(): Exception in A()
Handler in main(): Exception in A()
Handler in f(): Exception in f()
Returned value of f(): 1
```

Related References

- “The main() Function” on page 176
- “static Storage Class Specifier” on page 40
- Chapter 10, “Namespaces,” on page 229
- “Destructors” on page 320

Arguments of catch Blocks

C++ If you specify a class type for the argument of a catch block (the *exception_declaration*), the compiler uses a copy constructor to initialize that argument. If that argument does not have a name, the compiler initializes a temporary object and destroys it when the handler exits.

The ISO C++ specifications do not require the compiler to construct temporary objects in cases where they are redundant. The compiler takes advantage of this rule to create more efficient, optimized code. Take this into consideration when debugging your programs, especially for memory problems.

Related References

- “Temporary Objects” on page 327

Matching between Exceptions Thrown and Caught

C++ An argument in the catch argument of a handler matches an argument in the *assignment_expression* of the throw expression (throw argument) if any of the following conditions is met:

- The catch argument type matches the type of the thrown object.
- The catch argument is a public base class of the thrown class object.

- The catch specifies a pointer type, and the thrown object is a pointer type that can be converted to the pointer type of the catch argument by standard pointer conversion.

Note: If the type of the thrown object is **const** or **volatile**, the catch argument must also be a **const** or **volatile** for a match to occur. However, a **const**, **volatile**, or reference type catch argument can match a nonconstant, nonvolatile, or nonreference object type. A nonreference catch argument type matches a reference to an object of the same type.

Related References
- "Pointer Conversions" on page 152
- "Type Qualifiers" on page 71
- "References" on page 96
- "Special Exception Handling Functions" on page 383

Order of Catching

C++ If the compiler encounters an exception in a try block, it will try each handler in order of appearance.

If a catch block for objects of a base class precedes a catch block for objects of a class derived from that base class, the compiler issues a warning and continues to compile the program despite the unreachable code in the derived class handler.

A catch block of the form **catch(...)** must be the last catch block following a try block or an error occurs. This placement ensures that the **catch(...)** block does not prevent more specific catch blocks from catching exceptions intended for them.

If the run time cannot find a matching handler in the current scope, the run time will continue to find a matching handler in a dynamically surrounding try block. The following example demonstrates this:

```
#include <iostream>
using namespace std;

class E {
public:
  const char* error;
  E(const char* arg) : error(arg) { };
};

class F : public E {
public:
  F(const char* arg) : E(arg) { };
};

void f() {
  try {
    cout << "In try block of f()" << endl;
    throw E("Class E exception");
  }
  catch (F& e) {
    cout << "In handler of f()";
    cout << e.error << endl;
  }
};
int main() {
  try {
    cout << "In main" << endl;
    f();
  }
  catch (E& e) {
    cout << "In handler of main: ";
    cout << e.error << endl;
  };
  cout << "Resume execution in main" << endl;
}
```

The following is the output of the above example:

```
In main
In try block of f()
In handler of main: Class E exception
Resume execution in main
```

In function `f()`, the run time could not find a handler to handle the exception of type `E` thrown. The run time finds a matching handler in a dynamically surrounding try block: the try block in the `main()` function.

If the run time cannot find a matching handler in the program, it calls the `terminate()` function.

Related References
- "The try Keyword" on page 369
- "terminate()" on page 384

The throw Expression

C++ You use a *throw expression* to indicate that your program has encountered an exception.

Syntax – throw Expression

```
►►──throw──┬─────────────────────┬──────────────────────────────────────────►◄
           └─assignment_expression─┘
```

The type of *assignment_expression* cannot be an incomplete type, or a pointer to an incomplete type other than one of the following:
- **void***
- **const void***
- **volatile void***
- **const volatile void***

The *assignment_expression* is treated the same way as a function argument in a call or the operand of a return statement.

If the *assignment_expression* is a class object, the copy constructor and destructor of that object must be accessible. For example, you cannot throw a class object that has its copy constructor declared as private.

Related References
- "Incomplete Types" on page 77

Rethrowing an Exception

C++ If a catch block cannot handle the particular exception it has caught, you can rethrow the exception. The rethrow expression (**throw** without *assignment_expression*) causes the originally thrown object to be rethrown.

Because the exception has already been caught at the scope in which the rethrow expression occurs, it is rethrown out to the next dynamically enclosing try block. Therefore, it cannot be handled by catch blocks at the scope in which the rethrow expression occurred. Any catch blocks for the dynamically enclosing try block have an opportunity to catch the exception.

The following example demonstrates rethrowing an exception:

```
#include <iostream>
using namespace std;

struct E {
  const char* message;
  E() : message("Class E") { }
};

struct E1 : E {
  const char* message;
  E1() : message("Class E1") { }
};

struct E2 : E {
  const char* message;
  E2() : message("Class E2") { }
};

void f() {
  try {
    cout << "In try block of f()" << endl;
    cout << "Throwing exception of type E1" << endl;
    E1 myException;
    throw myException;
  }
  catch (E2& e) {
    cout << "In handler of f(), catch (E2& e)" << endl;
    cout << "Exception: " << e.message << endl;
    throw;
  }
  catch (E1& e) {
    cout << "In handler of f(), catch (E1& e)" << endl;
    cout << "Exception: " << e.message << endl;
    throw;
  }
  catch (E& e) {
    cout << "In handler of f(), catch (E& e)" << endl;
    cout << "Exception: " << e.message << endl;
    throw;
  }
}

int main() {
  try {
    cout << "In try block of main()" << endl;
    f();
  }
  catch (E2& e) {
    cout << "In handler of main(), catch (E2& e)" << endl;
    cout << "Exception: " << e.message << endl;
  }
  catch (...) {
    cout << "In handler of main(), catch (...)" << endl;
  }
}
```

The following is the output of the above example:

```
In try block of main()
In try block of f()
Throwing exception of type E1
In handler of f(), catch (E1& e)
Exception: Class E1
In handler of main(), catch (...)
```

The try block in the main() function calls function f(). The try block in function f() throws an object of type E1 named myException. The handler catch (E1 &e) catches myException. The handler then rethrows myException with the statement throw to the next dynamically enclosing try block: the try block in the main() function. The handler catch(...) catches myException.

Related References
- “The throw Expression” on page 377

Stack Unwinding

▶ C++ When an exception is thrown and control passes from a try block to a handler, the C++ run time calls destructors for all automatic objects constructed since the beginning of the try block. This process is called *stack unwinding*. The automatic objects are destroyed in reverse order of their construction. (Automatic objects are local objects that have been declared **auto** or **register**, or not declared **static** or **extern**. An automatic object x is deleted whenever the program exits the block in which x is declared.)

If an exception is thrown during construction of an object consisting of subobjects or array elements, destructors are only called for those subobjects or array elements successfully constructed before the exception was thrown. A destructor for a local static object will only be called if the object was successfully constructed.

If during stack unwinding a destructor throws an exception and that exception is not handled, the terminate() function is called. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct E {
  const char* message;
  E(const char* arg) : message(arg) { }
};

void my_terminate() {
  cout << "Call to my_terminate" << endl;
};

struct A {
  A() { cout << "In constructor of A" << endl; }
  ~A() {
    cout << "In destructor of A" << endl;
    throw E("Exception thrown in ~A()");
  }
};

struct B {
  B() { cout << "In constructor of B" << endl; }
  ~B() { cout << "In destructor of B" << endl; }
};

int main() {
  set_terminate(my_terminate);

  try {
    cout << "In try block" << endl;
    A a;
    B b;
    throw("Exception thrown in try block of main()");
  }
  catch (const char* e) {
    cout << "Exception: " << e << endl;
  }
  catch (...) {
    cout << "Some exception caught in main()" << endl;
  }

  cout << "Resume execution of main()" << endl;
}
```

The following is the output of the above example:

```
In try block
In constructor of A
In constructor of B
In destructor of B
In destructor of A
Call to my_terminate
```

In the try block, two automatic objects are created: `a` and `b`. The try block throws an exception of type **const char***. The handler `catch (const char* e)` catches this exception. The C++ run time unwinds the stack, calling the destructors for `a` and `b` in reverse order of their construction. The destructor for `a` throws an exception. Since there is no handler in the program that can handle this exception, the C++ run time calls `terminate()`. (The function `terminate()` calls the function specified as the argument to `set_terminate()`. In this example, `terminate()` has been specified to call `my_terminate()`.)

Related References

- “terminate()” on page 384
- “set_unexpected() and set_terminate()” on page 385

Exception Specifications

▶ C++ C++ provides a mechanism to ensure that a given function is limited to throwing only a specified list of exceptions. An exception specification at the beginning of any function acts as a guarantee to the function’s caller that the function will throw only the exceptions contained in the exception specification.

For example, a function:

```
void translate() throw(unknown_word,bad_grammar) { /* ... */ }
```

explicitly states that it will only throw exception objects whose types are `unknown_word` or `bad_grammar`, or any type derived from `unknown_word` or `bad_grammar`.

Syntax – Exception Specification

```
►►──throw──(──┬──────────────┬──)──────────────────────────────────────►◄
              └─type_id_list─┘
```

The *type_id_list* is a comma-separated list of types. In this list you cannot specify an incomplete type, a pointer or a reference to an incomplete type, other than a pointer to **void**, optionally qualified with **const** and/or **volatile**. You cannot define a type in an exception specification.

A function with no exception specification allows all exceptions. A function with an exception specification that has an empty *type_id_list*, **throw()**, does not allow any exceptions to be thrown.

An exception specification is not part of a function's type.

An exception specification may only appear at the end of a function declarator of a function, pointer to function, reference to function, pointer to member function declaration, or pointer to member function definition. An exception specification cannot appear in a **typedef** declaration. The following declarations demonstrate this:

```
  void f() throw(int);
  void (*g)() throw(int);
  void h(void i() throw(int));
  // typedef int (*j)() throw(int);  This is an error.
```

The compiler would not allow the last declaration, `typedef int (*j)() throw(int)`.

Suppose that class `A` is one of the types in the *type_id_list* of an exception specification of a function. That function may throw exception objects of class `A`, or any class publicly derived from class `A`. The following example demonstrates this:

```
class A { };
class B : public A { };
class C { };

void f(int i) throw (A) {
   switch (i) {
      case 0: throw A();
      case 1: throw B();
      default: throw C();
   }
}

void g(int i) throw (A*) {
   A* a = new A();
   B* b = new B();
   C* c = new C();
   switch (i) {
      case 0: throw a;
      case 1: throw b;
      default: throw c;
   }
}
```

Function `f()` can throw objects of types `A` or `B`. If the function tries to throw an object of type `C`, the compiler will call `unexpected()` because type `C` has not been specified in the function's exception specification, nor does it derive publicly from `A`. Similarly, function `g()` cannot throw pointers to objects of type `C`; the function may throw pointers of type `A` or pointers of objects that derive publicly from `A`.

A function that overrides a virtual function can only throw exceptions specified by the virtual function. The following example demonstrates this:

```
class A {
   public:
      virtual void f() throw (int, char);
};

class B : public A{
   public: void f() throw (int) { }
};

/* The following is not allowed. */
/*
   class C : public A {
      public: void f() { }
   };

   class D : public A {
      public: void f() throw (int, char, double) { }
   };
*/
```

The compiler allows `B::f()` because the member function may throw only exceptions of type **int**. The compiler would not allow `C::f()` because the member function may throw any kind of exception. The compiler would not allow `D::f()` because the member function can throw more types of exceptions (**int**, **char**, and **double**) than `A::f()`.

Suppose that you assign or initialize a pointer to function named `x` with a function or pointer to function named `y`. The pointer to function `x` can only throw exceptions specified by the exception specifications of `y`. The following example demonstrates this:

```
void (*f)();
void (*g)();
void (*h)() throw (int);

void i() {
   f = h;
//   h = g;  This is an error.
}
```

The compiler allows the assignment `f = h` because `f` can throw any kind of exception. The compiler would not allow the assignment `h = g` because `h` can only throw objects of type **int**, while `g` can throw any kind of exception.

Implicitly declared special member functions (default constructors, copy constructors, destructors, and copy assignment operators) have exception specifications. An implicitly declared special member function will have in its exception specification the types declared in the functions' exception specifications that the special function invokes. If any function that a special function invokes allows all exceptions, then that special function allows all exceptions. If all the functions that a special function invokes allow no exceptions, then that special function will allow no exceptions. The following example demonstrates this:

```
class A {
   public:
      A() throw (int);
      A(const A&) throw (float);
      ~A() throw();
};

class B {
   public:
      B() throw (char);
      B(const A&);
      ~B() throw();
};

class C : public B, public A { };
```

The following special functions in the above example have been implicitly declared:

```
C::C() throw (int, char);
C::C(const C&);   // Can throw any type of exception, including float
C::~C() throw();
```

The default constructor of C can throw exceptions of type **int** or **char**. The copy constructor of C can throw any kind of exception. The destructor of C cannot throw any exceptions.

Related References
- "Incomplete Types" on page 77
- "Function Declarations" on page 160
- "Pointers to Functions" on page 185
- Chapter 15, "Special Member Functions," on page 311
- "unexpected()"

Special Exception Handling Functions

C++ Not all thrown errors can be caught and successfully dealt with by a catch block. In some situations, the best way to handle an exception is to terminate the program. Two special library functions are implemented in C++ to process exceptions not properly handled by catch blocks or exceptions thrown outside of a valid try block. These functions are unexpected() and terminate().

unexpected()

C++ When a function with an exception specification throws an exception that is not listed in its exception specification, the C++ run time does the following:

1. The unexpected() function is called.
2. The unexpected() function calls the function pointed to by unexpected_handler. By default, unexpected_handler points to the function terminate().

You can replace the default value of unexpected_handler with the function set_unexpected().

Although unexpected() cannot return, it may throw (or rethrow) an exception. Suppose the exception specification of a function f() has been violated. If unexpected() throws an exception allowed by the exception specification of f(), then the C++ run time will search for another handler at the call of f(). The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct E {
  const char* message;
  E(const char* arg) : message(arg) { }
};

void my_unexpected() {
  cout << "Call to my_unexpected" << endl;
  throw E("Exception thrown from my_unexpected");
}
```

```
void f() throw(E) {
  cout << "In function f(), throw const char* object" << endl;
  throw("Exception, type const char*, thrown from f()");
}

int main() {
  set_unexpected(my_unexpected);
  try {
    f();
  }
  catch (E& e) {
    cout << "Exception in main(): " << e.message << endl;
  }
}
```

The following is the output of the above example:

```
In function f(), throw const char* object
Call to my_unexpected
Exception in main(): Exception thrown from my_unexpected
```

The `main()` function's try block calls function `f()`. Function `f()` throws an object of type **const char***. However the exception specification of `f()` allows only objects of type `E` to be thrown. The function `unexpected()` is called. The function `unexpected()` calls `my_unexpected()`. The function `my_unexpected()` throws an object of type `E`. Since `unexpected()` throws an object allowed by the exception specification of `f()`, the handler in the `main()` function may handle the exception.

If `unexpected()` did not throw (or rethrow) an object allowed by the exception specification of `f()`, then the C++ run time does one of two things:

- If the exception specification of `f()` included the class `std::bad_exception`, `unexpected()` will throw an object of type `std::bad_exception`, and the C++ run time will search for another handler at the call of `f()`.
- If the exception specification of `f()` did not include the class `std::bad_exception`, the function `terminate()` is called.

terminate()

C++ In some cases, the exception handling mechanism fails and a call to `void terminate()` is made. This `terminate()` call occurs in any of the following situations:

- The exception handling mechanism cannot find a handler for a thrown exception. The following are more specific cases of this:
  - During stack unwinding, a destructor throws an exception and that exception is not handled.
  - The expression that is thrown also throws an exception, and that exception is not handled.
  - The constructor or destructor of a nonlocal static object throws an exception, and the exception is not handled.
  - A function registered with `atexit()` throws an exception, and the exception is not handled. The following demonstrates this:

    ```
    extern "C" printf(char* ...);
    #include <exception>
    #include <cstdlib>
    using namespace std;

    void f() {
      printf("Function f()\n");
      throw "Exception thrown from f()";
    }
    ```

```
void g() { printf("Function g()\n"); }
void h() { printf("Function h()\n"); }

void my_terminate() {
  printf("Call to my_terminate\n");
  abort();
}

int main() {
  set_terminate(my_terminate);
  atexit(f);
  atexit(g);
  atexit(h);
  printf("In main\n");
}
```

The following is the output of the above example:

```
In main
Function h()
Function g()
Function f()
Call to my_terminate
```

To register a function with atexit(), you pass a parameter to atexit() a pointer to the function you want to register. At normal program termination, atexit() calls the functions you have registered with no arguments in reverse order. The atexit() function is in the <cstdlib> library.

- A throw expression without an operand tries to rethrow an exception, and no exception is presently being handled.
- A function f() throws an exception that violates its exception specification. The unexpected() function then throws an exception which violates the exception specification of f(), and the exception specification of f() did not include the class std::bad_exception.
- The default value of unexpected_handler is called.

The terminate() function calls the function pointed to by terminate_handler. By default, terminate_handler points to the function abort(), which exits from the program. You can replace the default value of terminate_handler with the function set_terminate().

A terminate function cannot return to its caller, either by using return or by throwing an exception.

set_unexpected() and set_terminate()

C++ The function unexpected(), when invoked, calls the function most recently supplied as an argument to set_unexpected(). If set_unexpected() has not yet been called, unexpected() calls terminate().

The function terminate(), when invoked, calls the function most recently supplied as an argument to set_terminate(). If set_terminate() has not yet been called, terminate() calls abort(), which ends the program.

You can use set_unexpected() and set_terminate() to register functions you define to be called by unexpected() and terminate(). The functions set_unexpected() and set_terminate() are included in the standard header files. Each of these functions has as its return type and its argument type a pointer to function with a **void** return type and no arguments. The pointer to function you supply as the argument becomes the function called by the corresponding special function: the argument to set_unexpected() becomes the function called by unexpected(), and the argument to set_terminate() becomes the function called by terminate().

Both set_unexpected() and set_terminate() return a pointer to the function that was previously called by their respective special functions (unexpected() and terminate()). By saving the return values, you can restore the original special functions later so that unexpected() and terminate() will once again call terminate() and abort().

If you use set_terminate() to register your own function, the function should no return to its caller but terminate execution of the program.

Example Using the Exception Handling Functions

C++ The following example shows the flow of control and special functions used in exception handling:

```
#include <iostream>
#include <exception>
using namespace std;

class X { };
class Y { };
class A { };

// pfv type is pointer to function returning void
typedef void (*pfv)();

void my_terminate() {
  cout << "Call to my terminate" << endl;
  abort();
}

void my_unexpected() {
  cout << "Call to my_unexpected()" << endl;
  throw;
}

void f() throw(X,Y, bad_exception) {
  throw A();
}

void g() throw(X,Y) {
  throw A();
}

int main()
{
  pfv old_term = set_terminate(my_terminate);
  pfv old_unex = set_unexpected(my_unexpected);
  try {
    cout << "In first try block" << endl;
    f();
  }
  catch(X) {
    cout << "Caught X" << endl;
  }
  catch(Y) {
    cout << "Caught Y" << endl;
  }
  catch (bad_exception& e1) {
    cout << "Caught bad_exception" << endl;
  }
  catch (...) {
    cout << "Caught some exception" << endl;
  }

  cout << endl;

  try {
    cout << "In second try block" << endl;
    g();
  }
  catch(X) {
    cout << "Caught X" << endl;
  }
  catch(Y) {
    cout << "Caught Y" << endl;
  }
  catch (bad_exception& e2) {
    cout << "Caught bad_exception" << endl;
  }
  catch (...) {
    cout << "Caught some exception" << endl;
  }
}
```

The following is the output of the above example:

```
In first try block
Call to my_unexpected()
Caught bad_exception

In second try block
Call to my_unexpected()
Call to my terminate
```

At run time, this program behaves as follows:

1. The call to `set_terminate()` assigns to `old_term` the address of the function last passed to `set_terminate()` when `set_terminate()` was previously called.
2. The call to `set_unexpected()` assigns to `old_unex` the address of the function last passed to `set_unexpected()` when `set_unexpected()` was previously called.
3. Within the first try block, function `f()` is called. Because `f()` throws an unexpected exception, a call to `unexpected()` is made. `unexpected()` in turn calls `my_unexpected()`, which prints a message to standard output. The function `my_unexpected()` tries to rethrow the exception of type `A`. Because class `A` has not been specified in the exception specification of function `f()`, `my_unexpected()` throws an exception of type `bad_exception`.
4. Because `bad_exception` has been specified in the exception specification of function `f()`, the handler `catch (bad_exception& e1)` is able to handle the exception.
5. Within the second try block, function `g()` is called. Because `g()` throws an unexpected exception, a call to `unexpected()` is made. The `unexpected()` throws an exception of type `bad_exception`. Because `bad_exception` has not been specified in the exception specification of `g()`, `unexpected()` calls `terminate()`, which calls the function `my_terminate()`.
6. `my_terminate()` displays a message then calls `abort()`, which terminates the program.

Note that the catch blocks following the second try block are not entered, because the exception was handled by `my_unexpected()` as an unexpected throw, not as a valid exception.

Appendix A. The IBM C Language Extensions

This appendix presents the IBM C extensions by category. The major categories are whether an extension is orthogonal or non-orthogonal to a base language. An orthogonal extension does not interfere with the base language. Orthogonal extensions are collectively enabled by compiling in one of the extended modes: `extended`, `extc89`, and `extc99`. The `extended` mode is based on C89.

Non-orthogonal extensions on the other hand may change the syntax or semantics of a base language feature. Therefore, each IBM C extension that is non-orthogonal to the base language or that conflicts with its GNU C implementation must be explicitly requested by an option.

The syntax for the positive and negative `langlvl` suboptions is:

```
-qlanglvl=lang_suboption
-qlanglvl=nolang_suboption
```

Options and suboptions are case-insensitive.

Orthogonal Extensions

The orthogonal IBM C extensions fall into three subgroupings: language features with individual option controls from previous releases, those that are C99 features, and those related to GNU C.

Existing IBM C Extensions with Individual Option Controls

Some existing language features that are orthogonal to C89 have individual positive and negative option controls. For backward compatibility, these compiler options and suboptions continue to be supported. Enabling a feature redundantly will not change its enabled state.

The IBM C language extensions with individual option controls

Language Extension	Compiler Option	Remarks
dollar sign in identifier	`-qdollar`	Accepted by all levels.
UCS	`-qlanglvl=ucs`	The negative setting, `-qlanglvl=noucs`, is ignored by STDC99 with an informational message.
digraph	`-qdigraph`	The negative setting, `-qnodigraph`, is ignored by STDC99 with an informational message.

IBM C Extensions: C99 Features as Extensions to C89

Most of the language features related to C99 are orthogonal to C89. The exception is the **restrict** keyword, which invades the user's variable name space. You can request the support explicitly by using the `-qkeyword=restrict` option.

C99 features as extensions to C89

Language Feature	Remarks
The **restrict** type qualifier	Defines a restricted pointer

C99 features as extensions to C89

Language Feature	Remarks
Variable length arrays	`-qlanglvl=c99vla`
Flexible array members	C99 allows a flexible array member only at the end of a struct. GNU C allows it anywhere in the structure.
Support for the complex data type	
The `long long int` type	
Support for hexadecimal floating-point constants	
Removal of implicit `int`	
Refined definition of integer division	Truncation toward zero
Universal character names	
Extended identifiers	Limit removed for internal and external names
Compound literals	
Designated initializers	
C++ style comments	
Removal of implicit function declaration	
Preprocessor arithmetic done in `intmax_t/uintmax_t`	
Mixed declarations and code	
New block scopes for selection and iteration statements	
Integer constant type rules	To accommodate the `long long int` type
Integer promotion rules	To accommodate the `long long int` type
vararg macros	Function-like macros with variable arguments
Trailing comma allowed in `enum` declaration	
Definition of the `_Bool` type	
Idempotent type qualifiers	Also known as ″duplicate type qualifiers″
Empty macro arguments	
Additional predefined macro names	
_Pragma preprocessing operator	
Standard pragmas	`#pragma STDC FP_CONTRACT#pragma STDC FENV_ACCESS#pragma STDC CX_LIMITED_RANGE`
`__func__` predefined identifier	
UTF-16, UTF-32 literals	

IBM C Extensions Related to GNU C

The IBM C compiler recognizes the following subset of the GNU C language extensions. The descriptive labels used in the following table are similar to those in the GNU C documentation.

The IBM C extensions related to GNU C

Language Feature	**Remarks**
Statements and Declarations in Expressions	
Locally Declared Labels	
Labels as Values	Including computed **goto** statements
Nested Functions	
Referring to a Type with **typeof**	The alternate spelling, **__typeof__**, is recommended.
Generalized Lvalues	
Double-Word Integers	
GNU C Complex Types	
GNU C Hexadecimal Float Constants	
Arrays of Length Zero	
Arrays of Variable Length	
Macros with a Variable Number of Arguments	Using an identifier in place of __VA_ARGS__
Non-Lvalue Arrays May Have Subscripts	
Non-Constant Initializers	
Compound Literals	
Cast to a Union Type	C-only
Declaring Attributes of Functions	
Function prototype overriding a nonprototype definition	
`__alignof__` to inquire about the alignment	
Specifying Attributes of Variables	
Specifying Attributes of Types	
Assembler Instructions with C Expression Operands	
Variables in Specified Registers	The compiler accepts the GNU syntax, but ignores the semantics.
Alternate Keywords	
`#warning`	
`#include_next`	

Non-Orthogonal Extensions

The non-orthogonal IBM C extensions fall into three subgroupings: language features from previous releases, those that are C99 features, and those related to GNU C.

Existing IBM C Extensions with Individual Option Controls

Strictly speaking, the IBM C language feature `upconv` is correctly classified as non-orthogonal. However, it is automatically enabled as part of the `extended` language level. This is the major difference between `extended` and `extc89`.

The non-orthogonal IBM C language extensions

Language Extension	Compiler Option	Remarks
`long long` literal	`-qlonglit`	Ignored by stdc99 with a warning.
`upconv`	`-qupconv`	Available by default at the `-qlanglvl=extended` language level.

IBM C Extensions: C99 Features as Extensions to C89

The non-orthogonal IBM C language extensions

Language Extension	Remarks
The `inline` keyword	Non-orthogonal to C89 and GNU C.
Flexible array members	C99 allows a flexible array member only at the end of a struct. GNU C allows it anywhere in the structure.

IBM C Extensions Related to GNU C

The non-orthogonal GNU C extension

Language Extension	Compiler Suboption and Remarks
Macros with a Variable Number of Arguments	Removing the trailing comma when no variable arguments are specified.

Extensions for AltiVec Programming Interface Support

▶ Linux ▶ Mac OS X

IBM C extensions for AltiVec vector programming

Language Extension	Compiler Suboption and Remarks
Vector programming language extensions	`-qaltivec` Keywords, macros, and pragmas to support the AltiVec programming model.

Appendix B. The IBM C++ Language Extensions

To maintain compatibility as a superset of C, IBM C++ implements features for compatibility at the C99 language level and with GNU C language extensions. In addition, IBM C++ supports a subset of the GNU extensions to C++. Like the IBM C language extensions, the C++ extensions have both orthogonal and non-orthogonal language features. An *orthogonal extension* is a feature that is added on top of a base without altering the behavior of the existing language features. A *non-orthogonal extension* is one that can change the semantics of existing constructs or can introduce syntax conflicting with the base.

This appendix presents the IBM C++ language extensions by category. The compiler option syntax for enabling and disabling an individual language feature is the same as for IBM C. All language extensions orthogonal to Standard C++ or C++98 are collectively enabled by compiling in the `extended` mode. Enabling a particular feature redundantly does not change its enabled state.

The compiler options are documented in detail in *XL C/C++ Compiler Reference*.

Orthogonal Extensions

The orthogonal IBM C++ extensions fall into three subgroupings: those that are related to C99 features to maintain compatibility with C, those related to GNU C, and those related to GNU C++.

IBM C++ Extensions for Compatibility with C99

IBM C++ adds the following C99 language features.

The IBM C++ extensions for compatibility with IBM C at the C99 language level

Language Feature	**Remarks**
The **restrict** type qualifier	Defines a restricted pointer or reference.
Variable length arrays	
Flexible array members	C99 allows a flexible array member only at the end of a struct. GNU C allows it anywhere in the structure.
Support for the complex data type	
Support for hexadecimal floating-point constants	
Universal character names	C++ code point range expanded to match that of C99
Compound literals	
vararg macros	Function-like macros with variable arguments
Trailing comma allowed in `enum` declaration	
Empty macro arguments	
Additional predefined macro names	
_Pragma preprocessing operator	
`__func__` predefined identifier	

The IBM C++ extensions for compatibility with IBM C at the C99 language level

Language Feature	Remarks
UTF-16, UTF-32 literals	

IBM C++ Extensions Related to GNU C

The orthogonal IBM C++ language extensions related to GNU C

Language Extension	Compiler Option and Remarks
Statements and Declarations in Expressions	-qlanglvl=extended
Locally Declared Labels	-qlanglvl=gnu_locallabel
Labels as Values	-qlanglvl=gnu_labelvalue
Computed **goto**	-qlanglvl=gnu_computedgoto
Referring to a Type with **typeof**	-qkeyword=__typeof__
GNU C Complex Types	-qlanglvl=c99complex, -qlanglvl=gnu_complex
GNU C Hexadecimal Float Constants	-qlanglvl=c99hexfloat
Array of Length Zero	-qlanglvl=compatzea
Arrays of Variable Length	-qlanglvl=c99vla
Macros with Variable Number of Arguments	-qlanglvl=varargsmacros, -qlanglvl=gnu_varargmacros
	Using an identifier in place of __VA_ARGS__.
Compound Literals	-qlanglvl=c99compoundliteral
Attributes for functions, variables, and types	-qkeyword=__attribute__
	Grouped together because all use the keyword **__attribute__**.
__alignof__ to inquire about the alignment	-qkeyword=__alignof__
Assembler Instructions with C Expression Operands	-qasm=gcc, -qkeyword=asm
Variables in specified registers	-qlanglvl=gnu_explicitregvar
	When the suboption is specified, the compiler accepts the GNU syntax but ignores the semantics.
Alternate Keywords	-qkeyword=inline -qkeyword=const -qkeyword=volatile -qkeyword=signed -qkeyword=__alignof__ -qkeyword=__asm__ -qkeyword=__inline__ -qkeyword=__const__ -qkeyword=__extension__ -qkeyword=__restrict__ -qkeyword=__signed__ -qkeyword=__typeof__ -qkeyword=__volatile__
	These options have been created for problematical keywords.
#assert, #unassert, #cpu, #machine, #system	-qlanglvl=gnu_assert
	Grouped together because all operate on assertions.
#warning	-qlanglvl=gnu_warning

The orthogonal IBM C++ language extensions related to GNU C

Language Extension	Compiler Option and Remarks
`#include_next`	`-qlanglvl=gnu_include_next`

IBM C++ Extensions Related to GNU C++

The orthogonal IBM C++ language extensions related to GNU C++

Language Extension	Compiler Option and Remarks
Variable Attribute `init_priority`	`-qlanglvl=__attribute__`
Locally Declared Labels	Supported only when declared in a block.

Non-Orthogonal Extensions

The non-orthogonal IBM C++ extensions fall into two subgroupings: those that are related to C99 features and those related to GNU C.

IBM C++ Extensions for Compatibility with C99

The non-orthogonal IBM C++ language extensions for compatibility with IBM C at the extended C99 language level

Language Extension	Remarks
The `inline` keyword	Non-orthogonal to Standard C++, C++98, C89, and GNU C.
Flexible array members	C99 allows a flexible array member only at the end of a struct. GNU C allows it anywhere in the structure. C++ provides partial support by allowing zero-extent arrays.

IBM C++ Extensions Related to GNU C

The non-orthogonal IBM C++ language extensions related to GNU C

Language Extension	Compiler Option and Remarks
typeof	`-qkeyword=typeof`
	Non-orthogonal because **typeof** is under the user namespace.
Macros with a Variable Number of Arguments	`-qlanglvl=varargsmacros`
	Removal of trailing comma when no variable macro arguments are specified.

Appendix C. AltiVec Data Types and Literals

This appendix presents the supported AltiVec data types and literals, which are supersets of those needed for strict conformance with the AltiVec specification.

Vector Data Types

The following table lists the supported AltiVec data types and the size and possible values for each type.

Type	Interpretation of content	Range of values
vector unsigned char	16 unsigned char	0..255
vector signed char	16 signed char	-128..127
vector bool char	16 unsigned char	0, 255
vector unsigned short	8 unsigned short	0..65535
vector unsigned short int		
vector signed short	8 signed short	-32768..32767
vector signed short int		
vector bool short	8 unsigned short	0, 65535
vector bool short int		
vector unsigned int	4 unsigned int	$0..2^{32}-1$
vector unsigned long§		
vector unsigned long int§		
vector signed int	4 signed int	$-2^{31}..2^{31}-1$
vector signed long§		
vector signed long int§		
vector bool int	4 unsigned int	$0, 2^{32}-1$
vector bool long§		
vector bool long int§		
vector float	4 float	IEEE-754 values
vector pixel	8 unsigned short	1/5/5/5 pixel

Note: § The AltiVec specification has deprecated **long** vector types.

The compiler considers any **long** vector data type compatible with the corresponding **int** vector type.

Vector Literals

The following table shows the supported vector literals and how the compiler interprets them to determine their values.

Syntax	Interpreted by the compiler as
(vector unsigned char)(unsigned int)	A set of 16 unsigned constants with a value specified by the integer constant expression.

Syntax	Interpreted by the compiler as
(vector unsigned char)(unsigned int, ..., unsigned int)	A set of 16 unsigned constants with a value specified by the 16 integer constant expressions.
(vector signed char)(int)	A set of 16 signed constants with a value specified by the integer constant expression.
(vector signed char)(int, ..., int)	A set of 16 signed constants with a value specified by the 16 integer constant expressions.
(vector bool char)(unsigned int)§	A set of 16 unsigned constants with a value specified by the integer constant expression.
(vector bool char)(unsigned int, ..., unsigned int)§	A set of 16 unsigned constants with a value specified by the 16 integer constant expressions.
(vector unsigned short)(unsigned int)	A set of 8 unsigned constants with a value specified by the integer constant expression.
(vector unsigned short)(unsigned int, ..., unsigned int)	A set of 8 unsigned constants with a value specified by the 8 integer constant expressions.
(vector signed short)(int)	A set of 8 signed constants with a value specified by the integer constant expression.
(vector signed short)(int, ..., int)	A set of 8 signed constants with a value specified by the 8 integer constant expressions.
(vector bool short)(unsigned int)§	A set of 8 unsigned constants with a value specified by the integer constant expression.
(vector bool short)(unsigned int, ..., unsigned int)§	A set of 8 unsigned constants with a value specified by the 8 integer constant expressions.
(vector unsigned int)(unsigned int)	A set of 4 unsigned constants with a value specified by the integer constant expression.
(vector unsigned int)(unsigned int, ..., unsigned int)	A set of 4 unsigned constants with a value specified by the 4 integer constant expressions.
(vector signed int)(int)	A set of 4 signed constants with a value specified by the integer constant expression.
(vector signed int)(int, ..., int)	A set of 4 signed constants with a value specified by the 4 integer constant expressions.
(vector bool int)(unsigned int)§	A set of 4 unsigned constants with a value specified by the integer constant expression.
(vector bool int)(unsigned int, ..., unsigned int)§	A set of 4 unsigned constants with a value specified by the 4 integer constant expressions.
(vector float)(float)	A set of 4 floating-point constants with a value specified by the floating-point constant expression.
(vector float)(float, float, float, float)	A set of 4 floating-point constants with a value specified by the 4 floating-point constant expressions.
(vector pixel)(unsigned int)§	A set of 8 unsigned constants with a value specified by the integer constant expression.
(vector pixel)(unsigned int, ..., unsigned int)§	A set of 8 unsigned constants with a value specified by the 8 integer constant expressions.

Note: § Denotes extension to the AltiVec programming interface.

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
B3/KB7/8200/MKM
8200 Warden Avenue
Markham, Ontario, Canada L6G 1C7

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 information 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.

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.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may 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, 2005. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming Interface Information

Programming interface information is intended to help you create application software using this program.

General-use programming interfaces allow the customer to write application software that obtains the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification, and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and Service Marks

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AIX
@server
IBM
POWER
PowerPC
pSeries
VisualAge

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

Linux is a 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 C (ANSI/ISO-IEC 9899–1990 [1992]). This standard has officially replaced American National Standard for Information Systems-Programming Language C (X3.159–1989) and is technically equivalent to the ANSI C standard. The compiler supports the changes adopted into the C Standard by ISO/IEC 9899:1990/Amendment 1:1994.
- The C language is 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)).

Index

Special characters

__align 20, 34
__cdecl 185
__func__ 19
__inline__ 20
__VA_ARGS__ 212, 214
_Pragma 226
- (subtraction operator) 135
- (unary minus operator) 121
-- (decrement operator) 120
-> (arrow operator) 112
, (comma operator) 146
:: (scope resolution operator) 107
! (logical negation operator) 121
!= (not equal to operator) 138
? : (conditional operators) 142
/ (division operator) 134
/= (compound assignment operator) 145
. (dot operator) 112
$ 13, 19
* (indirection operator) 123
* (multiplication operator) 134
*= (compound assignment operator) 145
\ continuation character 28, 210
\ escape character 13
[ ] (array subscript operator) 111
% (remainder) 135
> (greater than operator) 136
>> (right-shift operator) 136
>>= (compound assignment operator) 145
>= (greater than or equal to operator) 136
< (less than operator) 136
<< (left-shift operator) 136
<<= (compound assignment operator) 145
<= (less than or equal to operator) 136
| (bitwise inclusive OR operator) 140
| | (logical OR operator) 141
& (address operator) 122
& (bitwise AND operator) 139
& (reference declarator) 96
&& (label value operator) 126, 190
&& (logical AND operator) 140
&= (compound assignment operator) 145
# preprocessor directive character 210
# preprocessor operator 215
## (macro concatenation) 216
+ (addition operator) 135
+ (unary plus operator) 121
++ (increment operator) 120
+= (compound assignment operator) 145
= (simple assignment operator) 145
== (equal to operator) 138
^ (bitwise exclusive OR operator) 139
^= (compound assignment operator) 145
~ (bitwise negation operator) 121

A

aborting functions 385
abstract classes 306, 308
access rules
  base classes 291
  class types 253, 277
  friends 283
  members 276
  multiple access 298
  protected members 290
  virtual functions 307
access specifiers 264, 276, 287, 295
  in class derivations 291
accessibility 276, 298
addition operator (+) 135
address operator (&) 85, 122
  GNU C extension 190
  lvalue cast 132
aggregate types 44, 314
  initialization 57, 314
alias 96
alignment 32, 34
  bit fields 61
  structure members 56
  structures 32, 60
  structures and unions 34
  unions 60
  vector types 78
alignof operator 20, 123
allocation
  expressions 126
  functions 184
AltiVec data types 78
ambiguities
  base and derived member names 299
  base classes 297
  resolving 191, 300
  virtual function calls 306
AND operator, bitwise (&) 139
AND operator, logical (&&) 140
argc (argument count) 176
  example 176
arguments
  default 181
  evaluation 182
  macro 211
  main function 176
  of catch blocks 375
  passing 159, 177
  passing by reference 179
  passing by value 178
  trailing 212, 214
argv (argument vector) 176
  example 176
arithmetic conversions 71, 156
arithmetic types 49
  type compatibility 48
arrays
  array-to-pointer conversions 153
  as function parameter 41, 88
  declaration 41, 88, 89, 264
  description 87
  flexible array member 54, 56
  initialization 93
  initializing 91
  multidimensional 89
  subscripting operator 111
  type compatibility 88
  variable length 45, 90
  zero-extent 54, 57
ASCII character codes 13
asm 20, 75
assignment operator (=)
  compound 145
  pointers 84
  simple 145
associativity of operators 99
atexit function 384
auto storage class specifier 37

B

base classes
  abstract 308
  access rules 291
  ambiguities 297, 299
  direct 296
  indirect 286, 297
  initialization 316
  multiple access 298
  pointers to 289
  virtual 297, 301
base list 296
best viable function 248
binary expressions and operators 133
binding 97
  direct 97
  dynamic 302
  static 302
  virtual functions 302
bit fields 60
  as structure member 56
  type name 126
bitwise negation operator (~) 121
block statement 192
block visibility 3
bool 78
boolean
  conversions 151
  literals 21
  variables 49
brackets
  bracketed form 93
break statement 202

C

- candidate functions 237, 248
- case label 195
- cast expressions 30, 79, 131
  - complex to real 71
  - union type 132
- catch blocks 172, 369, 371
  - argument matching 375
  - order of catching 376
- char type specifier 50
- character
  - data types 50
  - literals 26
  - multibyte 16, 28
- character set
  - extended 16
  - source 12
- class members
  - access operators 112
  - access rules 276
  - class member list 263
  - declaration 264
  - initialization 316
  - order of allocation 264
- class templates
  - declaration and definition 345
  - distinction from *template class* 343
  - explicit specialization 361
  - member functions 346
  - static data members 345
- classes 255
  - abstract 308
  - access rules 276
  - aggregate 254
  - base 287
  - base list 287
  - class objects 35, 36
  - class specifiers 253
  - class templates 343
  - declarations 253
    - incomplete 257, 264
  - derived 287
  - friends 278
  - inheritance 285
  - keywords 253
  - local 260
  - member functions 264
  - member lists 263
  - member scope 267
  - nested 258, 281
  - overview 253
  - polymorphic 253
  - scope of names 256
  - static members 272
  - this pointer 269
  - using declaration 292
  - virtual 297, 302
- comma 146
  - in enumerator list 67
- comments 16
- compatible types
  - across source files 48
  - arithmetic types 48
  - arrays 88
  - in conditional expressions 142
  - vector types 78
- complex type 70, 120
- composite types 48
  - across source files 48
- compound
  - assignment 145
  - expression 146
  - literal 21, 28, 108
  - statement 192
  - types 54
- computed goto 126, 190, 207
- concatenation
  - macros 216
  - multibyte characters 28
  - u-literals, U-literals 15
- conditional compilation directives 220
  - elif preprocessor directive 222
  - else preprocessor directive 223
  - endif preprocessor directive 223
  - examples 224
  - if preprocessor directive 222
  - ifdef preprocessor directive 222
  - ifndef preprocessor directive 223
- conditional expression (? :) 142, 146
- const 73, 78
  - casting away constness 180
  - function attribute 165
  - member functions 266
  - object 103
  - placement in type name 45
  - qualifier 71
  - vs. #define 210
- const_cast 116, 180
- constant expressions 67, 106
- constant initializers 263
- constructors 312
  - converting 156, 330
  - copy 332
  - exception handling 379
  - exception thrown in constructor 372
  - initialization
    - explicit 314
  - initializer list 172
  - nontrivial 313, 321
  - overview 311
  - trivial 313, 321
- continuation character 28, 210
- continue statement 203
- conversion
  - constructors 330
  - function 331
  - implicit conversion sequences 249
- conversion sequence
  - ellipsis 249
  - implicit 249
  - standard 249
  - user-defined 249
- conversions
  - argument expressions 177
  - arithmetic 156
  - array-to-pointer 153
  - boolean 151
  - cast 131
  - derived-to-base 153
  - explicit keyword 156
  - floating-point 152
  - function arguments 154
  - function-to-pointer 153
  - integral 151
- conversions *(continued)*
  - lvalue-to-rvalue 103, 150, 249
  - pointer 152
  - pointer to derived class 301
  - pointer to member 154
  - qualification 154
  - references 153
  - standard 150
  - user-defined 328
  - void pointer 152
- copy assignment operators 333
- copy constructors 332
- covariant virtual functions 304
- CPLUSPLUS macro 220
- cv-qualifier 56, 71, 82
  - in function definition 172
  - in parameter type specification 238
  - syntax 71

D

- data members
  - description 264
  - scope 267
  - static 273
- DATE macro 219
- deallocation
  - expressions 130
  - functions 184
- decimal integer literals 22
- declarations
  - classes 253, 257
  - description 31
  - friend specifier in member list 278
  - friends 283
  - pointers to members 268
  - resolving ambiguous statements 191
  - syntax 31, 44
  - unsubscripted arrays 90
  - vector types 78
- declarative region 1
- declarators
  - description 81
  - reference 96
- decrement operator (−−) 120
- default
  - clause 195, 196
  - label 196
- default constructor 313
- define preprocessor directive 210
- defined unary operator 222
- definitions
  - description 31
  - macro 210
  - member function 265
  - tentative 35
- delete operator 130
- dependent names 365
- dereferencing operator 123
- derivation 287
  - array type 88
  - public, protected, private 291
- derived classes
  - catch block 376
  - construction order 319
  - pointers to 289

- designated initializer
  - aggregate types 57, 93
  - union 63
- designator 57, 93
  - designation 57, 93
  - designator list 57, 93
  - union 63
- destructors 320
  - exception handling 379
  - exception thrown in destructor 372
  - overview 311
  - pseudo 112, 322
- digraph 21
- direct base class 296
- division operator (/) 134
- do statement 200
- dollar sign 13, 19
- dot operator 112
- double type specifier 51
- downcast 118
- dynamic binding 302
- dynamic_cast 117

E

- EBCDIC character codes 13
- elaborated type specifier 256
- elif preprocessor directive 222
- ellipsis
  - conversion sequence 249
  - in function declaration 161
  - in function definition 174
  - in macro argument list 212
- else
  - preprocessor directive 223
  - statement 193
- enclosing class 265, 280
- endif preprocessor directive 223
- entry point
  - program 176
- enum
  - keyword 67, 69
- enumerations 67
  - compatibility 48, 67
  - declaration 67
  - initialization 69
  - trailing comma 67
- enumerator 67
- equal to operator (==) 138
- error preprocessor directive 217
- escape character \ 13
- escape sequence 13
  - alarm \a 13
  - backslash \\ 13
  - backspace \b 13
  - carriage return \r 13
  - double quotation mark \″ 13
  - form feed \f 13
  - horizontal tab \t 13
  - new-line \n 13
  - question mark \? 13
  - single quotation mark \′ 13
  - vertical tab \v 13
- exception handling 369
  - argument matching 375
  - catch blocks 371
    - arguments 375
- exception handling *(continued)*
  - constructors 379
  - destructors 379
  - example, C++ 386
  - exception objects 369
  - function try blocks 369
  - handlers 369, 371
  - order of catching 376
  - rethrowing exceptions 377
  - set_terminate 385
  - set_unexpected 385
  - special functions 383
  - stack unwinding 379
  - terminate function 384
  - throw expressions 371, 377
  - try blocks 369
  - try exceptions 372
  - unexpected function 383
- exceptions
  - declaration 371
  - function try block handlers 372
  - specification 172, 380
- exclusive OR operator, bitwise (^) 139
- explicit
  - function specifier 96
  - instantiation, templates 357
  - keyword 156, 330
  - specializations, templates 358, 359
  - type conversions 131
- exponent 24
- expressions
  - allocation 126
  - assignment 144
  - binary 133
  - cast 131
  - comma 146
  - conditional 142
  - deallocation 130
  - description 99
  - full 99
  - integer constant 106
  - new initializer 129
  - parenthesized 106
  - pointer to member 142
  - primary 104
  - resolving ambiguous statements 191
  - statement 191
  - throw 148, 377
  - unary 119
- extern inline
  - keyword 38, 186
- extern storage class specifier 7, 9, 38, 186
  - implicit declaration 109
  - with function pointers 185
  - with template declaration 355, 357
  - with variable length arrays 90

F

- file inclusion 218, 219
- FILE macro 219
- file scope data declarations
  - unsubscripted arrays 90
- flexible array member 57
- float type specifier 51
- floating types 49, 71
- floating-point
  - constant 25
  - conversion 152
  - literal 23
  - promotion 149
- for statement 200
- free store 323
  - delete operator 130
  - new operator 126
- friend
  - access rules 283
  - implicit conversion of pointers 292
  - member functions 264
  - nested classes 281
  - relationships with classes when templates are involved 346
  - scope 280
  - specifier 278
  - virtual functions 305
- function attributes 163
- function designator 103
- function specifier
  - explicit 156, 330
- function templates
  - explicit specialization 360
- function try blocks 172, 369
  - handlers 372
- function-like macro 211
- functions 159
  - allocation 184
  - arguments 109, 159, 160
    - conversions 154
  - block 159
  - body 159
  - C++ enhancements 159
  - calling 177
  - calls 109
    - as lvalue 103
  - class templates 346
  - conversion function 331
  - deallocation 184
  - declaration 159, 160
    - C++ 162
    - examples 169
    - exception specification 161
    - multiple 162
    - parameter names 163
  - default arguments 181
    - evaluation 182
    - restrictions 181
  - definition 159, 160, 170
    - constructor initializer list 172
    - declarator 171
    - examples 174
    - exception specification 172
    - return type 172
    - scope 171
    - storage class specifier 171
    - try block 172
    - type specifiers 172
  - exception handling 383
  - exception specification 380
  - friends 278
  - function call operator 159
  - function templates 347
  - function-to-pointer conversions 153
  - inline 38, 186, 265

- functions *(continued)*
  - library functions 159
  - main 176
  - name 160, 172
    - diagnostic 19
  - overloading 237
  - parameters 77, 109, 160, 177
  - pointers to 185
  - polymorphic 286
  - predefined identifier 19
  - prototype 159, 161
  - return statements 204
  - return type 159, 172, 183, 184
  - return value 159, 183
  - signature 159
  - specifiable attributes 163
  - specifiers 96, 186
  - template function
    - template argument deduction 348
  - type name 45
  - virtual 266, 302, 306
- fundamental type 49

G

- global variable 3, 7
  - uninitialized 38
- goto statement 206
  - computed goto 207
  - restrictions 206
- greater than operator (>) 136
- greater than or equal to operator (>=) 136

H

- handlers 371
- hexadecimal
  - floating constants 25
- hexadecimal integer literals 23
- hidden names 254, 256

I

- identifiers 18, 104
  - case sensitivity 19
  - id-expression 82, 105
  - labels 189
  - linkage 7
  - name space 4
  - predefined 19
  - reserved 18, 20
  - special characters 13, 19
- if
  - preprocessor directive 222
  - statement 193
- ifdef preprocessor directive 222
- ifndef preprocessor directive 223
- imaginary type 70
- imaginary unit 71
- implementation dependency
  - allocation of floating-point types 51
  - allocation of integral types 52
- implicit conversion 149
  - boolean 151
  - lvalue 103
- implicit conversion *(continued)*
  - pointer to derived class 289, 292
  - pointers to base class 290
  - types 149
- implicit instantiation
  - templates 355
- implicit int 175
- include preprocessor directive 218
- include_next preprocessor directive 219
- inclusive OR operator, bitwise (|) 140
- incomplete type 77, 88
  - as structure member 54, 56
  - class declaration 257
- increment operator (++) 120
- indentation of code 210
- indirect base class 286, 297
- indirection operator (*) 79, 85, 123
- information hiding 1, 3, 263, 290
- inheritance
  - multiple 286, 296
  - overview 285
- initialization
  - aggregate types 57
  - auto object 37
  - base classes 316
  - class members 316
  - extern object 38
  - order of 33
  - references 153
  - register object 40
  - static data members 275
  - static object 29, 41
  - union member 63
  - vector types 79
- initializer lists 29, 30, 79, 82, 172, 316
- initializers 82
  - aggregate types 57, 93
  - enumerations 69
  - unions 63
  - vector types 29, 30, 79
- inline
  - function specifier 96, 186
  - functions 186, 265
  - keyword 20, 38
- integer
  - constant expressions 67, 106
  - conversion 151
  - conversions 151
  - data types 52
  - implicit int 172
  - literals 22
  - promotion 149

K

- keywords 20
  - exception handling 369
  - language extension 20
  - template 337, 366
  - underscore characters 20

L

- label
  - as values 190
  - implicit declaration 3
- label *(continued)*
  - in switch statement 196
  - locally declared 190
  - statement 189
- language extension x, 20
  - C
    - non-orthogonal 391
    - orthogonal 389
  - C++
    - non-orthogonal 395
    - orthogonal 393
  - C99 389
  - GNU C x, 389
- left-shift operator (<<) 136
- less than operator (<) 136
- less than or equal to operator (<=) 136
- LINE macro 219
- line preprocessor directive 224
- linkage 1, 6
  - auto storage class specifier 37
  - const cv-qualifier 73
  - extern storage class specifier 9, 39
  - external 7, 109
  - in function definition 171, 172
  - inline functions 186
  - inline member functions 266
  - internal 6, 40
  - language 8
  - multiple function declarations 163
  - none 7
  - register storage class specifier 40
  - specifications 8
  - static storage class specifier 42
  - weak symbols 34
  - with function pointers 185
- linking to non-C++ programs 8
- literals 21
  - boolean 21
  - character 26
  - compound 21, 28, 108
  - floating-point 23
  - integer 22
    - data types 22
    - decimal 22
    - hexadecimal 23
    - octal 23
  - string 27
  - Unicode 14
- local
  - classes 260
  - type names 261
- logical operators
  - ! (logical negation) 121
  - | | (logical OR) 141
  - && (logical AND) 140
- long double type specifier 51
- long long type specifier 49, 52, 78
- long type specifier 52, 78
- lvalues 71, 103, 104
  - casting 131
  - conversions 103, 150, 249

M

- macro
  - definition 210
    - typeof operator 126

- macro *(continued)*
  - function-like 211
  - invocation 211
  - object-like 211
  - variable argument 212, 214
- main function 176
  - arguments 176
  - example 176
- member functions
  - const and volatile 266
  - definition 265
  - friend 264
  - special 266
  - static 275
  - this pointer 269, 307
- member lists 254, 263
- members
  - access 276
  - access control 295
  - class member access operators 112
  - data 264
  - pointers to 142, 268
  - protected 290
  - scope 267
  - static 259, 272
  - virtual functions 266
- modifiable lvalue 103, 144, 145
- modulo operator (%) 135
- multibyte character 16
  - concatenation 28
- multicharacter literal 26
- multidimensional arrays 89
- multiple
  - access 298
  - inheritance 286, 296
- multiplication operator (*) 134
- mutable storage class specifier 39

N

- name binding 365
- name hiding 5, 107
  - accessible base class 301
  - ambiguities 299
- name space
  - class names 256
  - context 5
  - of identifiers 4
- names
  - conflicts 4
  - hidden 107, 254, 256
  - local type 261
  - mangling 9
  - resolution 3, 292, 300
- namespaces 229
  - alias 229, 230
  - declaring 229
  - defining 229
  - explicit access 235
  - extending 230
  - friends 233
  - member definitions 233
  - namespace scope object
    - exception thrown in
      - constructor 372
  - overloading 231
  - unnamed 231
- namespaces *(continued)*
  - user-defined 2
  - using declaration 234
  - using directive 234
- narrow character literal 26
- nested classes
  - friend scope 281
  - scope 258
- new operator
  - default arguments 181
  - description 126
  - initializer expression 129
  - placement syntax 128, 323, 324
  - set_new_handler function 129
- non-orthogonal language extensions
  - C 391
- nonorthogonal language extensions
  - C++ 395
- not equal to operator (!=) 138
- null
  - character \0 28
  - pointer 85
  - pointer constants 153
  - preprocessor directive 225
  - statement 207
- number sign (#)
  - preprocessor directive character 210
  - preprocessor operator 215

O

- object-like macro 211
- objects 103
  - class
    - declarations 254
  - description 35
  - lifetime 1
  - namespace scope
    - exception thrown in
      - constructor 372
  - restrict-qualified pointer 74
  - static
    - exception thrown in
      - destructor 372
- octal integer literals 23
- one's complement operator (~) 121
- operator functions 239
- operators 11
  - - (subtraction) 135
  - -- (decrement) 120
  - -> (arrow) 112
  - ->* (pointer to member) 142
  - , (comma) 146
  - :: (scope resolution) 107
  - ! (logical negation) 121
  - != (not equal to) 138
  - ? : (conditional) 142
  - / (division) 134
  - . (dot) 112
  - .* (pointer to member) 142
  - () (function call) 109, 159
  - * (indirection) 123
  - * (multiplication) 134
  - – (unary minus) 121
  - [] (array subscripting) 111
  - % (remainder) 135
  - > (greater than) 136
- operators *(continued)*
  - >> (right- shift) 136
  - >= (greater than or equal to) 136
  - < (less than) 136
  - << (left- shift) 136
  - <= (less than or equal to) 136
  - | (bitwise inclusive OR) 140
  - || (logical OR) 141
  - & (address) 122
  - & (bitwise AND) 139
  - && (logical AND) 140
  - + (addition) 135
  - ++ (increment) 120
  - = (simple assignment) 145
  - == (equal to) 138
  - ^ (bitwise exclusive OR) 139
  - alternative representations 12, 21
  - assignment 144
    - copy assignment 333
  - associativity 99
  - binary 133
  - bitwise negation operator (~) 121
  - compound assignment 145
  - const_cast 116
  - defined 222
  - delete 130, 325
  - dynamic_cast 117
  - equality 138
  - expressions 108
  - new 126, 323
  - operators 108
  - overloading 239, 264
    - binary 243
    - unary 241
  - pointer to member 142, 269
  - precedence 99
    - examples 102
    - type names 44
  - preprocessor
    - # 215
    - ## 216
    - pragma 226
  - reinterpret_cast 114
  - relational 136
  - scope resolution 288, 299, 305
  - sizeof 124
  - static_cast 114
  - typeid 112
  - typeof 125
  - unary 119
  - unary plus operator (+) 121
- OR operator, logical (||) 141
- orthogonal language extensions
  - C 389
  - C++ 393
- overload resolution 247, 302
  - resolving addresses of overloaded functions 250
- overloading
  - delete operator 325
  - description 237
  - function templates 353
  - functions 237, 293
    - restrictions 238
  - new operator 323
  - operators 239, 253
    - assignment 243

- overloading *(continued)*
  - operators *(continued)*
    - binary 243
    - class member access 247
    - decrement 241
    - function call 245
    - increment 241
    - subscripting 246
    - unary 241
- overriding virtual functions 306
  - covariant virtual function 304

P

- packed
  - structure member 60
  - variable attribute 34
- parenthesized expressions 44, 106
- pass by reference 96, 179
- pass by value 178
- pixel 78
- placement syntax 128, 324
- pointer to member
  - conversions 154
  - declarations 268
  - operators 142, 269
- pointers
  - compatible 84
  - const 73
  - conversions 152, 301
  - cv-qualified 84
  - description 83
  - generic 152
  - null 85
  - pointer arithmetic 79, 86
  - restrict-qualified 74
  - this 269
  - to functions 185
  - to members 142, 268
  - type-qualified 84
  - vector types 79
  - void* 152
- polymorphism
  - polymorphic classes 253, 303
  - polymorphic functions 286
- postfix
  - ++ and -- 120
  - expression 108
  - operator 108
- pound sign (#)
  - preprocessor directive character 210
  - preprocessor operator 215
- pragma operator 119, 226
- pragmas
  - _Pragma 226
  - preprocessor directive 226
  - standard 226
- precedence of operators 99
- predefined identifier 19
- predefined macros
  - __IBMC__ 220
  - __IBMCPP__ 220
  - CPLUSPLUS 220
  - DATE 219
  - FILE 219
  - LINE 219
  - STDC 220
- predefined macros *(continued)*
  - STDC_HOSTED 220
  - STDC_VERSION 220
  - TIME 220
- prefix
  - ++ and -- 120
  - hexadecimal floating constants 25
  - hexadecimal integer literals 23
  - octal integer literals 23
- preprocessor directives 209
  - conditional compilation 220
  - preprocessing overview 209
  - special character 210
  - warning 217
- preprocessor operator
  - _Pragma 226
  - # 215
  - ## 216
- primary expressions 104
- promotions
  - function argument values 177
  - integral and floating-point 149
- prototype 161
- pseudo-destructors 112
- punctuators 11
  - alternative representations 12, 21
- pure specifier 263, 266, 305, 308
- pure virtual functions 308

Q

- qualification conversions 154
- qualified name 107, 259
- qualifiers
  - const 71, 172
  - in parameter type specification 238
  - restrict 74
  - volatile 71, 74, 172

R

- references
  - as return types 184
  - binding 97
  - conversions 153
  - declarator 122
  - description 96
  - initialization 97
- register storage class specifier 39
- reinterpret_cast 114
- remainder operator (%) 135
- restrict 74
  - in parameter type specification 238
- return statement 183, 204
  - value 205
- return type
  - reference as 184
  - size_t 124
- right-shift operator (>>) 136
- RTTI support 112, 113, 117
- rvalues 103

S

- scalar types 44, 83
- scope 1
- scope *(continued)*
  - class 4
  - class names 256
  - description 1
  - enclosing and nested 3
  - friends 280
  - function 3
  - function prototype 3
  - global 3
  - global namespace 4
  - identifiers 4
  - local (block) 3
  - local classes 260
  - macro names 215
  - member 267
  - nested classes 258
- scope resolution operator
  - ambiguous base classes 299
  - description 107
  - inheritance 288
  - virtual functions 305
- sequence point 99, 147
- set_new_handler function 129
- set_terminate function 385
- set_unexpected function 383, 385
- shift operators << and >> 136
- short type specifier 52
- side effect 74, 99
- signed type specifiers
  - char 50
  - int 52
  - long 52
  - long long 52
- simple type specifiers 49
  - char 50
  - wchar_t 50
- size_t 124
- sizeof operator 124
  - with variable length arrays 91
- space character 210
- special characters 13
- special member functions 266
- specifiers
  - access control 291
  - inline 186
  - pure 266
  - storage class 36
- splice preprocessor directive ## 216
- stack unwinding 379
- Standard C ix
- Standard C++ ix
- standard type conversions 149, 150
- statement expression 193
- statements 189
  - block 192
  - break 202
  - compound 193
  - continue 203
  - do 200
  - expressions 191
  - for 200
  - goto 206
  - if 193
  - labels 189
  - null 207
  - resolving ambiguities 191
  - return 183, 204

statements *(continued)*
- selection 193, 195
- switch 195
- while 199

static 78
- binding 302
- data members 273
- in array declaration 41, 88
- initialization of data members 275
- member functions 275
- members 259, 272
- storage class specifier 40
  - linkage 42
- with variable length arrays 90

static storage class specifier 7
static_cast 114
STDC macro 220
STDC_HOSTED macro 220
STDC_VERSION macro 220
storage class specifiers 36, 171
- auto 37
- extern 38
- mutable 39
- register 39
- static 40

storage duration 1
- auto storage class specifier 37
- extern storage class specifier 38
- register storage class specifier 40
- static 40
  - exception thrown in destructor 372

string
- literal 27
- terminator 28

stringize preprocessor directive # 215
struct type specifier 55
structures 54, 255
- alignment 34, 60
- as base class 291
- as class type 253, 254
- compatibility 48, 55, 60
- embedded pragma directives 60
- flexible array member 54, 56
- identifier (tag) 55
- initialization 57
- members 56
  - alignment 56
  - incomplete types 56
  - layout in memory 57
  - packed 60
  - padding 56
  - zero-extent array 54
- name spaces within 5
- nesting 60
- packed 56
- type attributes 45
- unnamed members 58

subscript declarator
- in arrays 89

subscripting operator 87, 111
- in type name 45

subtraction operator (−) 135
suffix
- floating-point literals 23
- hexadecimal floating constants 25
- integer literal constants 22

switch statement 195

T

tags
- enumeration 67
- structure 55, 56
- union 64

template arguments 340
- deduction 348
- deduction, non-type 353
- deduction, type 351
- non-type 340
- template 342
- type 340

template keyword 366
templates
- arguments
  - non-type 340
  - type 340
- class
  - declaration and definition 345
  - distinction from *template class* 343
  - explicit specialization 361
  - member functions 346
  - static data members 345
- declaration 337
- dependent names 365
- explicit specializations 358, 359
  - class members 360
  - declaration 358
  - definition and declaration 359
  - function templates 360
- function
  - argument deduction 353
  - overloading 353
  - partial ordering 354
- function templates 347
  - type template argument deduction 351
- instantiation 337, 355, 358
  - explicit 357
  - forward declaration 355, 357
  - implicit 355
- name binding 365
- parameters 338
  - default arguments 339
  - non-type 338
  - template 339
  - type 338
- partial specialization 362
  - matching 364
  - parameter and argument lists 364
- point of definition 365
- point of instantiation 365
- relationship between classes and their friends 346
- scope 359
- specialization 337, 355, 358

temporary objects 327, 375
tentative definition 35
terminate function 369, 371, 376, 379, 383, 384
- set_terminate 385

this pointer 73, 269, 307
throw expressions 148, 369, 377
- argument matching 375

throw expressions *(continued)*
- rethrowing exceptions 377
- within nested try blocks 371

TIME macro 220
tokens 11, 209
- alternative representations for operators and punctuators 12

translation unit 1
trigraph sequences 15
truncation
- integer division 134

try blocks 369
- in function definition 172
- nested 371

try keyword 369
type attributes 45
type name 44
- local 261
- qualified 107, 259
- typename keyword 366
- typeof operator 125

type qualifiers
- const 71, 73
- const and volatile 82
- in function parameters 238
- in structure member definition 56
- restrict 71, 74
- volatile 71

type specifier 44
- (long) double 51
- AltiVec data types 78
- char 50
- class types 253
- complex 70
- elaborated 256
- enumeration 67
- float 51
- imaginary 70
- in function definition 171
- int 52
- long 52
- long long 52
- overriding 33
- short 52
- simple 49
- unsigned 52
- wchar_t 50, 52

typedef specifier 42
- and type compatibility 48
- class declaration 261
- local type names 261
- pointers to members 269
- qualified type name 259
- vector types 78
- with variable length arrays 91

typeid operator 112
typename keyword 366
typeof operator 20, 125
types
- aggregates 44
- class 253
- compatible 48
- composite 48
- compound 54
- conversions 131
- enumerated 67
- floating 71

types *(continued)*
- scalar 44
- variably modified 89
- vector 78

U

u-literal, U-literal 14
unary expressions 119
unary operators 119
- label value 126
- minus (–) 121
- plus (+) 121

undef preprocessor directive 215
underscore character 18, 20
unexpected function 369, 383, 384
- set_unexpected 385

Unicode 14
unions 63
- alignment 60
- as class type 253, 254
- cast to union type 132
- compatibility 48, 60, 63
- designated initializer 57
- embedded pragma directives 60
- initialization 63
- nesting 60
- specifier 64
- type attributes 45
- unnamed members 58

universal character name 14, 18, 27
unnamed namespaces 231
unsigned type specifiers
- char 50
- int 52
- long 52
- long long 52
- short 52

unsubscripted arrays
- description 88, 90

user-defined conversions 328
using declarations 234, 292, 300
- changing member access 295
- overloading member functions 293

using directive 234
UTF-16, UTF-32 14

V

variable attributes 32
variable length array 77, 90, 206
- as function parameter 91, 177, 178, 248
- as structure member 56
- as union members 63
- sizeof 106, 124
- type name 45

variably modified types 56, 63, 89, 90, 197
- size evaluation 177

vector types 78, 125, 397
- in typedef declarations 42
- literals 29, 397

virtual
- base classes 287, 297, 301
- function specifier 96

virtual functions 266, 302
- access 307
- ambiguous calls 306
- overriding 306
- pure specifier 308

visibility 1, 5
- block 3
- class members 278

void 53
- argument type 174
- in function definition 172, 174
- pointer 152

volatile
- member functions 266
- qualifier 71, 74

W

warning preprocessor directive 217
wchar_t type specifier 27, 50, 52
weak symbol 34
while statement 199
white space 11, 16, 209, 210, 216
wide characters
- literals 26

wide string literal 28

Z

zero-extent array 57

Program Number: 5724-K77

SC09-7945-01