z/OS

XL C/C++ Language Reference

Version 2 Release 1.1

SC14-7308-01

Note
Before using this information and the product it supports, read the information in "Notices" on page 661.

This edition applies to Version 2 Release 1 of z/OS (5650-ZOS) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

About this information

This information describes the syntax, semantics, and IBM® z/OS® XL C/C++ implementation of the C and C++ programming languages. Although the XL C and XL C++ compilers conform to the specifications maintained by the ISO standards for the C and C++ programming languages, the compilers also incorporate many extensions to the core languages. These extensions have been implemented with the aims of enhancing usability in specific operating environments, supporting compatibility with other compilers, and supporting new hardware capabilities. For example, on the z/OS platform, language constructs have been added to provide support for data types that are specific to the IBM System z® environment.

Note: As of z/OS V1R7, IBM z/OS C/C++ compiler has been rebranded to IBM z/OS XL C/C++.

Who should read this information

This information is a reference for users who already have experience programming applications in C or C++. Users new to C or C++ can still use this information to find language and features unique to XL C/C++; however, this reference does not aim to teach programming concepts nor to promote specific programming practices.

How to use this information

Unless indicated otherwise, all of the text in this reference pertains to both C and C++ languages. Where there are differences between languages, these are indicated through qualifying text and other graphical elements (see below for the conventions used).

While this information covers both standard and implementation-specific features, it does not include the following topic:

- Standard C and C++ library functions and headers. For standard C/C++ library information, refer to the *Standard C++ Library Reference*.

How this information is organized

This information is organized to loosely follow the structure of the ISO standard language specifications and topics are grouped into similar headings.

- Chapters 3 through 10 discuss language elements that are common to both C and C++, including scope and linkage, lexical elements, data types, declarations, declarators, type conversions, expressions, operators, statements, and functions. Throughout these chapters, both standard features and extensions are discussed.
- Chapters 11 through 18 discuss standard C++ features exclusively, including classes, overloading, inheritance, templates, and exception handling.
- Chapters 19 through 22 discuss directives to the preprocessor and macros that are predefined by the compiler.
- Chapters 23 through 25 discuss the compatibility and conformance on the z/OS platform.

- The last chapters discuss implementation-defined behavior, accessibility, and notices.

Conventions

Typographical conventions

The following table shows the typographical conventions used in the information.

Table 1. Typographical conventions

Typeface	Indicates	Example
bold	Lowercase commands, executable names, compiler options, and directives.	The compiler provides basic invocation commands, **xlc** and **xlC** (**xlc++**), along with several other compiler invocation commands to support various C language levels and compilation environments.
italics	Parameters or variables whose actual names or values are to be supplied by the user. Italics are also used to introduce new terms.	Make sure that you update the *size* parameter if you return more than the *size* requested.
underlining	The default setting of a parameter of a compiler option or directive.	nomaf | maf
`monospace`	Programming keywords and library functions, compiler builtins, examples of program code, command strings, or user-defined names.	To compile and optimize myprogram.c, enter: `xlc myprogram.c -O3`.

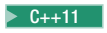
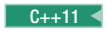
Qualifying elements (icons)

Most features described in this information apply to both C and C++ languages. In descriptions of language elements where a feature is exclusive to one language, or where functionality differs between languages, this information uses icons to delineate segments of text as follows:

Table 2. Qualifying elements

Qualifier/Icon	Meaning
C only, or C only begins   C only ends	The text describes a feature that is supported in the C language only; or describes behavior that is specific to the C language.
C++ only, or C++ only begins   C++ only ends	The text describes a feature that is supported in the C++ language only; or describes behavior that is specific to the C++ language.

Table 2. Qualifying elements (continued)

Qualifier/Icon	Meaning
IBM extension, or IBM extension begins   IBM extension ends	The text describes a feature that is an IBM extension to the standard language specifications.
C11, or C11 begins   C11 ends	The text describes a feature that is introduced into standard C as part of C11.
C++11, or C++11 begins   C++11 ends	The text describes a feature that is introduced into standard C++ as part of C++11.
z/OS only   	The text describes a feature that is supported only on the z/OS implementation of the compilers.
C++ and C11, or C++ and C11 begin   C++ and C11 end	The text describes a feature that is supported by both C++ and C11 standards.

Syntax diagrams

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

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

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

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

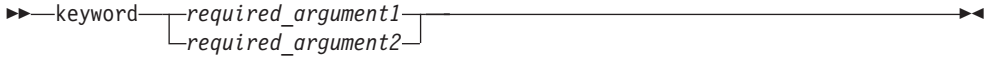
  The ▸—— symbol indicates that a command, directive, or statement is continued from the previous line.

  The ——▸◂ symbol indicates the end of a command, directive, or statement.

  Fragments, which are diagrams of syntactical units other than complete commands, directives, or statements, start with the |—— symbol and end with the ——| symbol.
- Required items are shown on the horizontal line (the main path):

- Optional items are shown below the main path:

- If you can choose from two or more items, they are shown vertically, in a stack.
  If you *must* choose one of the items, one item of the stack is shown on the main path.



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

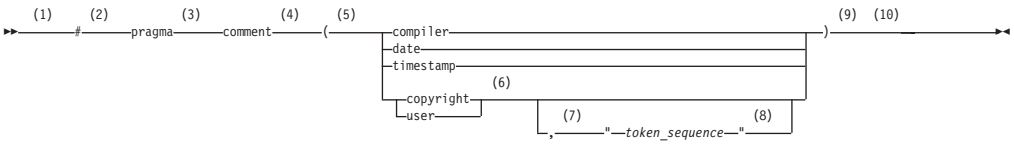
- An arrow returning to the left above the main line (a repeat arrow) indicates that you can make more than one choice from the stacked items or repeat an item. The separator character, if it is other than a blank, is also indicated:

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

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

Sample syntax diagram

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



Notes:

1 This is the start of the syntax diagram.

2 The symbol # must appear first.

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

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

5 An opening parenthesis must be present.

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

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

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

9 A closing parenthesis is required.

10 This is the end of the syntax diagram.

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

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

Examples in this information

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

z/OS XL C/C++ and related documents

z/OS XL C/C++ documents address a variety of application development tasks and are provided in multiple formats.

For a summary of the information contained in z/OS XL C/C++ documents see "z/OS XL C/C++ and related documents" in *z/OS XL C/C++ User's Guide*.

Softcopy documents

The z/OS XL C/C++ documents are supplied in PDF and IBM BookMaster® formats on the following CD: *z/OS Collection*, SK3T-4269. They are also available at http://www.ibm.com/software/awdtools/czos/library/.

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

You can also browse the documents on the World Wide Web by visiting the z/OS library at http://www.ibm.com/systems/z/os/zos/bkserv/.

Note: For further information on viewing and printing softcopy documents and using IBM BookManager®, see *z/OS Information Roadmap*.

Softcopy examples

For information on the labelling used to identify examples that are available as softcopy files, see "Softcopy examples" in *z/OS XL C/C++ User's Guide*.

z/OS XL C/C++ on the World Wide Web

Additional information on z/OS XL C/C++ is available on the World Wide Web on the z/OS XL C/C++ home page at: http://www.ibm.com/software/awdtools/czos/.

This page contains late-breaking information about the z/OS XL C/C++ product, including the compiler, the C/C++ libraries, and utilities. There are links to other useful information, such as the z/OS XL C/C++ information library and the libraries of other z/OS elements that are available on the Web. The z/OS XL C/C++ home page also contains links to other related Web sites.

Technical support

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

If you cannot find what you need, you can e-mail:

compinfo@ca.ibm.com

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

For information about boosting performance, productivity and portability, visit the C/C++ Cafe at: http://www-949.ibm.com/software/rational/cafe/community/ccpp.

How to send your comments

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

compinfo@ca.ibm.com

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

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.

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 is potentially 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 the 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 is 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.

The distinctions between the different types of scopes are discussed in “Scope” on page 2. The different types of linkages are discussed in “Program linkage” on page 7.

Related reference:

“Storage class specifiers” on page 48

Chapter 9, “Namespaces (C++ only),” on page 317

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.

Table 3. Kinds of scope

C	C++
block	local
function	function
Function prototype	Function prototype
file	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 scope.

Related reference:

Chapter 9, “Namespaces (C++ only),” on page 317

Block/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 immediately enclosing 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 reference:

"Block statements" on page 199

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 reference:

"Labeled statements" on page 197

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 reference:

"Function declarations" on page 219

File/global scope

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

C++ *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 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.

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. C++

Related reference:

Chapter 9, "Namespaces (C++ only)," on page 317

"Internal linkage" on page 7
"The extern storage class specifier" on page 51

Examples of scope in C

The following example declares the variable x on line 1, which is different from the x it declares on line 2. The declared variable on line 2 has function prototype scope and is visible only up to the closing parenthesis of the prototype declaration. The variable x declared on line 1 resumes visibility after the end of the prototype declaration.

```
1   int x = 4;             /* variable x defined with file scope */
2   long myfunc(int x, long y); /* variable x has function       */
3                               /* prototype scope               */
4   int main(void)
5   {
6      /* . . . */
7   }
```

The following program illustrates blocks, nesting, and scope. The example shows two kinds of scope: file and block. The main function prints the values 1, 2, 3, 0, 3, 2, 1 on separate lines. Each instance of i represents a different variable.

```
            #include <stdio.h>
            int i = 1;                          /* i defined at file scope */

            int main(int argc, char * argv[])
┌──────     {
1
1              printf("%d\n", i);               /* Prints 1 */
1
1  ┌───       {
1  2             int i = 2, j = 3;            /* i and j defined at block scope */
1  2                                         /* global definition of i is hidden */
1  2             printf("%d\n%d\n", i, j);    /* Prints 2, 3 */
1  2
1  2  ┌──       {
1  2  3             int i = 0;  /* i is redefined in a nested block */
1  2  3                         /* previous definitions of i are hidden */
1  2  3             printf("%d\n%d\n", i, j); /* Prints 0, 3 */
1  2  └──       }
1  2
1  2             printf("%d\n", i);           /* Prints 2 */
1  2
1  └───       }
1
1              printf("%d\n", i);               /* Prints 1 */
1
1              return 0;
1
└──────    }
```

Class scope (C++ only)

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 reference:

Chapter 11, "Classes (C++ only)," on page 347

"Scope of class names" on page 351

"Member scope" on page 361

"Friend scope" on page 377

"Access control of base class members" on page 387

"Scope resolution operator :: (C++ only)" on page 148

Namespaces of identifiers

Namespaces 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 *namespace* as used here applies to C as well as C++ and does not refer to the C++ namespace language feature. The compiler sets up *namespaces* to distinguish among identifiers referring to different kinds of entities. Identical identifiers in different namespaces 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 namespace. The syntactic context of an identifier within a program lets the compiler resolve its namespace without ambiguity.

Within each of the following four namespaces, the identifiers must be unique:

- *Tags* of the following 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 namespace using enclosed program blocks.

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

```
int get_item()
{
   struct student        /* structure tag */
   {
      char student[20];  /* structure member */
      int section;
      int id;
   } student;            /* structure variable */

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

The compiler interprets each occurrence of `student` by its context in the program: when `student` appears after the keyword `struct`, it is a structure tag; when it appears in the block defining the `student` type, it is a structure member variable; when it appears at the end of the structure definition, it declares a structure variable; and when it appears after the `goto` statement, it is a label.

Name hiding (C++ only)

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 namespace `B` through the use of a 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;
}
```

See 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 reference:

Chapter 11, “Classes (C++ only),” on page 347

“Member functions” on page 359

“Member scope” on page 361

Chapter 9, “Namespaces (C++ only),” on page 317

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:

- “Internal linkage” : identifiers can only be seen within a translation unit.
- “External linkage” on page 8 : identifiers can be seen (and referred to) in other translation units.
- “No linkage” on page 8: identifiers 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.

Related reference:

“The static storage class specifier” on page 49

“The extern storage class specifier” on page 51

“Function storage class specifiers” on page 225

“Type qualifiers” on page 85

Anonymous unions

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` C++11 or `constexpr` C++11 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

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 entities will also have external linkage:

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

C++

Related reference:

"The _Export qualifier (C++ only)" on page 125

"The _Export function specifier (C++ only)" on page 233

No linkage

The following kinds of identifiers have no linkage:

- Names that have neither external nor internal linkage
- Names declared in local scopes (with exceptions of 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 C++ template names C++

You cannot use a name with no linkage to declare an entity with linkage. For example, you cannot use the name of a structure 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;
}
```

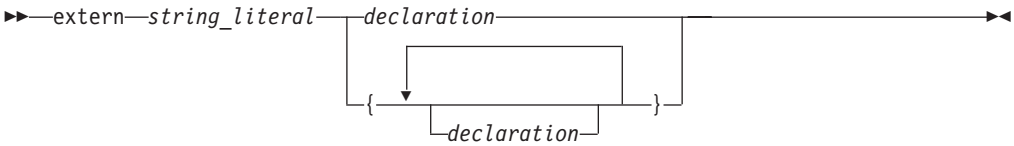
The compiler will not allow the declaration of `a1` with external linkage. Structure `A` has no linkage. The compiler will not allow the declaration of `a2` with external linkage. The `typedef` name `myA` has no linkage because `A` has no linkage.

Language linkage (C++ only)

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*.

Linkage specification syntax



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');
}
```

CCNX02J

```
// This example illustrates linkage specifications.

extern "C" int printf(const char*,...);

int main(void)
{
      printf("hello\n");
}
```

Here the *string_literal* `"C"` tells the compiler that the routine `printf(const char*,...)` is a C function.

Note: This example is not guaranteed to work on all platforms. The only safe way to declare a C function in a C++ program is to include the appropriate header. In this example you would substitute the line of code with `extern` with the following line:

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

Name mangling (C++ only)

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. Name mangling is commonly used to facilitate the overloading feature and visibility within different scopes. The compiler generates function names with an encoding of the types of the function arguments when the module is compiled. 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.

Related reference:

"The extern storage class specifier" on page 51

"The extern storage class specifier" on page 226

"#pragma linkage (C only)" on page 566

"The __cdecl function specifier (C++ only)" on page 231

Chapter 2. Lexical elements

A *lexical element* refers to a character or groupings of characters that might legally appear in a source file. This chapter contains discussions of the basic lexical elements and conventions of the C and C++ programming languages.

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.

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.

There are the following different types of tokens:

- "Keywords"
- "Identifiers" on page 16
- "Literals" on page 19
- "Punctuators and operators" on page 30

Keywords

Keywords are identifiers reserved by the language for special use. Although you can use them for preprocessor macro names, it is considered poor programming style. Only the exact spelling of keywords is reserved. For example, `auto` is reserved but `AUTO` is not.

Keywords for the C and C++ languages

Table 4. C and C++ keywords

auto[1]	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern[2]	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Notes:

1. C++11 In C++11, the keyword `auto` is no longer used as a storage class specifier. Instead, it is used as a type specifier, which can deduce the type of an `auto` variable from the type of its initializer expression.
2. The keyword `extern` was previously used as a storage specifier or as part of a linkage specification. The C++11 standard adds a third usage to use this keyword to specify explicit instantiation declarations. C++11

C

Keywords for the C language only

Standard C at the C99 and C11 levels also reserves the following keywords:

Table 5. C99 and C11 keywords

_Atomic[1]	inline[3]
_Bool	_Noreturn[1]
_Complex	_Static_assert[1]
_Generic[1]	restrict[4]
_Imaginary[2]	

Notes:

1. C11 These keywords are introduced due to the C11 language level. C11
2. The keyword `_Imaginary` is reserved for possible future use. For complex number functionality, use `_Complex`; see Complex literals (C only) for details.
3. The keyword `inline` is only recognized under compilation with **c99** and above, or with the LANGLVL(STDC99) or LANGLVL(EXTC99) options.
4. The keyword `restrict` is only recognized under compilation with **c99** and above, or with the LANGLVL(STDC99) or LANGLVL(EXTC99) options.

C

C++

Keywords for the C++ language only

The C++ language also reserves the following keywords:

Table 6. C++ keywords

bool	dynamic_cast	new	this
catch	decltype		throw
class	explicit	operator	true
char16_t[1]	export	private	try
char32_t[1]	false	protected	typeid
const_cast	friend	public	typename
constexpr	inline	reinterpret_cast	using
delete	mutable	static_assert	virtual
	namespace	static_cast	wchar_t
		template	

Note:

1. C++11 These keywords are reserved only at the C++11 language level. C++11

C++

Keywords for language extensions (IBM extension)

In addition to standard language keywords, the z/OS XL C/C++ compiler reserves the following keywords for use in language extensions:

Table 7. Keywords for C and C++ language extensions

asm	__imag__ (C only)	__signed
__asm	__inline__	__static_assert[1]
__asm__	_Noreturn[2]	typeof (C only)
__attribute__	__real__ (C only)	__typeof__
__attribute	__restrict	vector[3]
__complex__ (C only)	__restrict__	__vector[3]
__const__	__signed__	__volatile (C++ only)
__extension__		__volatile__ (C++ only)

Notes:

1. C++11 `__static_assert` is a keyword for C language extension for compatibility with the C++11 standard.
2. C++ `_Noreturn` is a keyword for C++ language extension for compatibility with the C11 standard. C++
3. These keywords are recognized only when the VECTOR compiler option is in effect.

C++ The z/OS XL C/C++ compiler reserves the following keywords as language extensions for compatibility with C99.

Table 8. Keywords for C++ language extensions related to C99

restrict

C++

z/OS z/OS XL C/C++ additionally reserves the following for use as extensions:

Table 9. Keywords for C/C++ language extensions on z/OS

C	C++	
		__callback
_Packed	__cdecl	__ptr32
__packed	_Export	__ptr64
		__far[1]

Note:

1. Recognized only when the METAL compiler option is in effect, which is currently only supported by z/OS XL C.

z/OS XL C/C++ also reserves the following keywords for future use in both C and C++:

Table 10. Reserved keywords for future use

__alignof__
__extension__
__label__
C++ _Pragma

z/OS

More detailed information regarding the compilation contexts in which extension keywords are valid is provided in the sections that describe each keyword.

Identifiers

Identifiers provide names for the following language elements:

- Functions
- Objects
- Labels
- Function parameters
- Macros and macro parameters
- Type definitions
- Enumerated types and enumerators
- Structure 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:

Characters in identifiers

The first character in an identifier must be a letter or the _ (underscore) character; however, beginning identifiers with an underscore is considered poor programming style.

The compiler distinguishes between uppercase and lowercase letters in identifiers. For example, `PROFIT` and `profit` represent different identifiers. If you specify a lowercase `a` as part of an identifier name, you cannot substitute an uppercase `A` in its place; you must use the lowercase letter.

Note: C If the names have external linkage, and you do not specify the LONGNAME compiler option, names are truncated to eight characters and uppercased in the object file. For example, `STOCKONHOLD` and `stockonhold` will both refer to the same object. For more information on external name mapping, see "External identifiers (z/OS only)."

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. C++ In C++, you must compile with the LANGLVL(UCS) option for universal character name support.

External identifiers (z/OS only)

By default, external names in C object modules, and external names without C++ linkage in C++ object modules, are formatted as follows:

- All characters are converted to uppercase.

- Names longer than 8 characters are truncated to 8 characters.
- Each underscore character is converted to an at sign (@).

For example, if you compile the following C program:

```
int test_name[4] = { 4, 8, 9, 10 };
int test_namesum;

int main(void) {
  int i;
  test_namesum = 0;

  for (i = 0; i < 4; i++)
    test_namesum += test_name[i];
  printf("sum is %d\n", test_namesum);
}
```

The C compiler displays the following message:

```
ERROR  CCN3244 ./sum.c:2   External name TEST_NAM cannot be redefined.
```

The compiler changes the external names `test_namesum` and `test_name` to uppercase and truncates them to 8 characters. If you specify the CHECKOUT compile-time option, the compiler will generate two informational messages to this effect. Because the truncated names are now the same, the compiler produces an error message and terminates the compilation.

To avoid this problem, you can do either of the following:

- Map long external names in the source code to 8 or less characters that you specify, by using the **#pragma map** directive. For example:

  ```
  #pragma map(verylongname,"sname")
  ```

- Compile with the LONGNAME compiler option, and use the binder to produce a program object in a PDSE, or use the prelinker. This allows up to 1024 characters in external names, mixed-case characters, and preserves the underscore character. For more information on the binder, prelinker, and LONGNAME compile-time option, see the *z/OS XL C/C++ User's Guide*.

IBM-provided functions have names that begin with IBM, CEE, and PLI. In order to prevent conflicts between runtime functions and user-defined names, the compiler changes all `static` or `extern` variable names that begin with IBM, CEE, and PLI in your source program to IB$, CE$, and PL$, respectively, in the object module. If you are using interlanguage calls, avoid using these prefixes altogether. The compiler of the calling or called language may or may not change these prefixes in the same manner as the z/OS XL C/C++ compiler does.

To call an external program or access an external variable that begins with IBM, CEE, or PLI, use the **#pragma map** preprocessor directive. The following is an example of **#pragma map** that forces an external name to be `IBMENTRY`:

```
#pragma map(ibmentry,"IBMENTRY")
```

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 a single underscore are reserved as identifiers with file scope in both the ordinary and tag namespaces.

C++ Identifiers that begin with a single underscore are reserved in the global namespace.

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.

The __func__ predefined identifier

The C99 predefined identifier `__func__` makes a function name available for use within the function. C++ The z/OS XL C/C++ compiler supports this feature as an IBM extension. C++ 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.

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

C++ 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>`.

For debugging purposes, you can explicitly use the `__func__` identifier to return the name of the function in which it appears. For example:

```
#include <stdio.h>

void myfunc(void)    {
         printf("%s\n",__func__);
         printf("size of __func__ = %d\n", sizeof(__func__));
}

int main() {
     myfunc();
}
```

The output of the program is:

```
myfunc
size of __func__ = 7
```

When the `assert` macro is used inside a function definition, the macro adds the name of the enclosing function on the standard error stream.

Related reference:

"Identifier expressions (C++ only)" on page 145

"The Unicode standard" on page 35

"Keywords" on page 13

"#pragma map" on page 570

"#pragma longname/nolongname" on page 569
"Function declarations and definitions" on page 219
Variables in specified registers (IBM extension)
"Inline assembly statements (IBM extension)" on page 215
"Command-line arguments" on page 248

Literals

The term *literal constant*, or *literal*, refers to a value that occurs in a program and cannot be changed. C The C language uses the term *constant* in place of the noun *literal* C . 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.

There are the following different types of literals:

- "Integer literals"
- "Boolean literals" on page 23
- "Floating-point literals" on page 23
- z/OS "Fixed-point decimal literals" on page 27
- "Character literals" on page 27
- "String literals" on page 28
- "Pointer literal (C++11)" on page 30

Integer literals

C++11

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

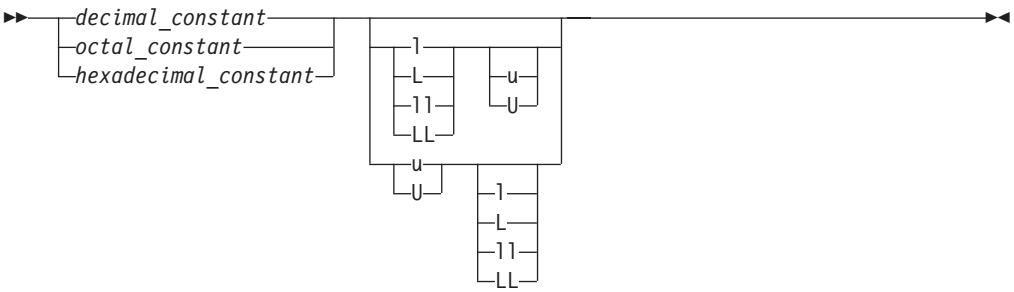
C++11

Integer literals are numbers that do not have a decimal point or an exponential part. They can be represented as:

- Decimal integer literals
- Hexadecimal integer literals
- Octal integer literals

An integer literal might have a prefix that specifies its base, or a suffix that specifies its type.

Integer literal syntax



The `long long` features

There are two `long long` features:

- the C99 `long long` feature
- the non-C99 `long long` feature

Note: The syntax of integer literals is the same for both of the `long long` features.
IBM Both of the two features have the corresponding extension parts:

- the C99 `long long` feature with the associated IBM extensions
- the non-C99 IBM `long long` extension

IBM

Types of integer literals outside of C99 and C++11

The following table lists the integer literals and shows the possible data types when the C99 `long long` feature is not enabled.

Table 11. Types of integer literals outside of C99 and C++11[1]

Representation	Suffix	Possible data types					
		int	unsigned int	long int	unsigned long int	IBM long long int	IBM unsigned long long int
Decimal	None	+		+	+[2]		
Octal, Hex	None	+	+	+	+		
All	`u` or `U`		+		+		
Decimal	`l` or `L`			+	+		
Octal, Hex	`l` or `L`			+	+		
All	Both `u` or `U` and `l` or `L`				+		
Decimal	`ll` or `LL`					+	+
Octal, Hex	`ll` or `LL`					+	+

Table 11. Types of integer literals outside of C99 and C++11[1] (continued)

Representation	Suffix	Possible data types					
All	Both `u` or `U` and `ll` or `LL`						+

Notes:

1. When none of the `long long` features are enabled, types of integer literals include all the types in this table except the last two columns.
2. IBM The `unsigned long int` type is not required here in the C++98 and C++03 standards. The C++ compiler includes the type in the implementation for compatibility purposes only.

Types of integer literals in C99 and C++11

When both the C99 and non-C99 `long long` features are disabled, integer literals that have one of the following suffixes cause a severe compile-time error:

- `ll` or `LL`
- Both `u` or `U` and `ll` or `LL`

IBM A decimal literal without a `u` or `U` in the suffix is represented by the `unsigned long long int` type if both of the following conditions are satisfied. In this case, the compiler generates a message to indicate that the value of the literal is too large for any signed integer type.

- The value of the literal can fit into the `unsigned long long int` type.
- The value cannot fit into any of the possible data types that are indicated in the following table.

IBM

C++11 IBM To strictly conform to the C++11 standard, the compiler introduces the extended integer safe behavior to ensure that a signed value never becomes an unsigned value after a promotion. After you enable this behavior, if a decimal integer literal that does not have a suffix containing `u` or `U` cannot be represented by the `long long int` type, the compiler issues an error message to indicate that the value of the literal is out of range. The extended integer safe behavior is the only difference between the C99 `long long` feature with the associated IBM extensions and the C99 `long long` feature. IBM C++11

The following table lists the integer literals and shows the possible data types when the C99 `long long` feature is enabled.

Table 12. Types of integer literals in C99 and C++11

Representation	Suffix	Possible data types					
		int	unsigned int	long int	unsigned long int	long long int	unsigned long long int
Decimal	None	+		+			
Octal, Hex	None	+	+	+	+		
All	`u` or `U`		+		+		
Decimal	`l` or `L`			+			
Octal, Hex	`l` or `L`			+	+		

Table 12. Types of integer literals in C99 and C++11 (continued)

Representation	Suffix	Possible data types					
All	Both `u` or `U` and `l` or `L`				+		
Decimal	`ll` or `LL`					+	+1
Octal, Hex	`ll` or `LL`					+	+
All	Both `u` or `U` and `ll` or `LL`						+

Note:

1. C++11 IBM The compiler does not support this type if the extended integer safe behavior is enabled.
2. z/OS In 32-bit mode, an unsuffixed decimal constant of type `signed long long` is given the type `signed long` in 64-bit mode when the constant is less than ULLONG_MAX.

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.

Decimal integer literal syntax

See the following examples of decimal literals:

```
485976
5
```

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

```
-433132211
+20
```

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.

Hexadecimal integer literal syntax

See the following 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.

Octal integer literal syntax

See the following examples of octal integer literals:

```
0
0125
034673
03245
```

Related reference:

"Integral types" on page 54

"Integral conversions" on page 130

"Integral and floating-point promotions" on page 135

"C++11 compatibility" on page 640

Boolean literals

C At the C99 level, C defines `true` and `false` as macros in the header file `stdbool.h`.

C++ There are only two Boolean literals: `true` and `false`.

Related reference:

"Boolean types" on page 55

"Boolean conversions" on page 130

Floating-point literals

Floating-point literals are numbers that have a decimal point or an exponential part. They can be represented as:

- Real literals
  - Binary floating-point literals

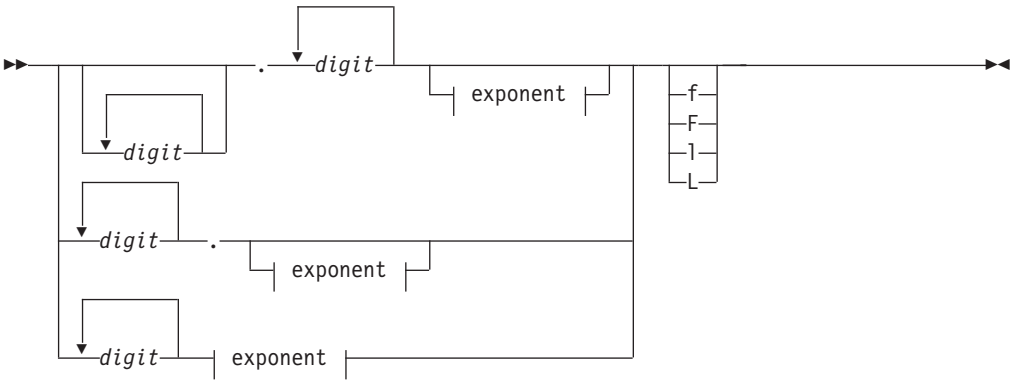
– C Hexadecimal floating-point literals (C only)
- Complex literals

Binary floating-point literals

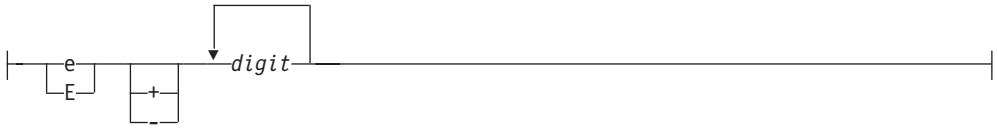
A real binary floating-point constant consists of the following:
- An integral part
- A decimal point
- A fractional part
- An exponent part
- An optional suffix

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.

Binary floating-point literal syntax



Exponent:



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

floating-point constant	Value
7.321E-3	0.007321
3.2E+4	32000
0.5e-6	0.0000005
0.45	0.45
6.e10	60000000000

Hexadecimal floating-point literals (C only)

Real hexadecimal floating constants, which are a C99 feature, consist of the following parts.

- a hexadecimal prefix
- a significant part
- a binary exponent part
- an optional suffix

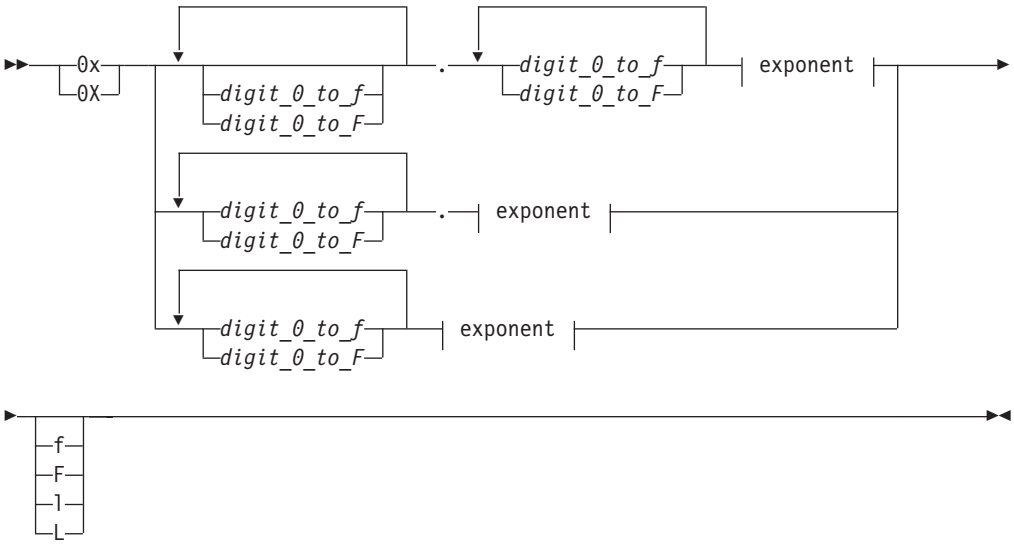
The significant part represents a rational number and is composed of the following:

- 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:

Hexadecimal floating-point literal syntax



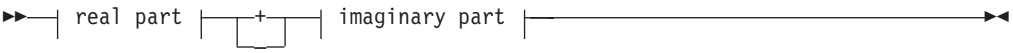
Exponent:

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`. 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

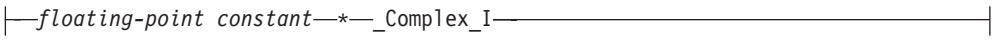
Complex literals, which were introduced in the C99 standard, are constructed in two parts: the real part, and the imaginary part.

Complex literal syntax



Real part:

Imaginary part:



floating-point constant can be specified as a hexadecimal floating-point literal (including optional suffixes), in any of the formats described in the previous sections.

`_Complex_I` is a macro defined in the `complex.h` header file, representing the imaginary unit *i*, the square root of -1.

For example, the declaration:

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

initializes the complex variable `varComplex` to a value of 2.0 + 2.0*i*.

Related reference:

"Floating-point types" on page 56

"Floating-point conversions" on page 130

"Unary expressions" on page 151

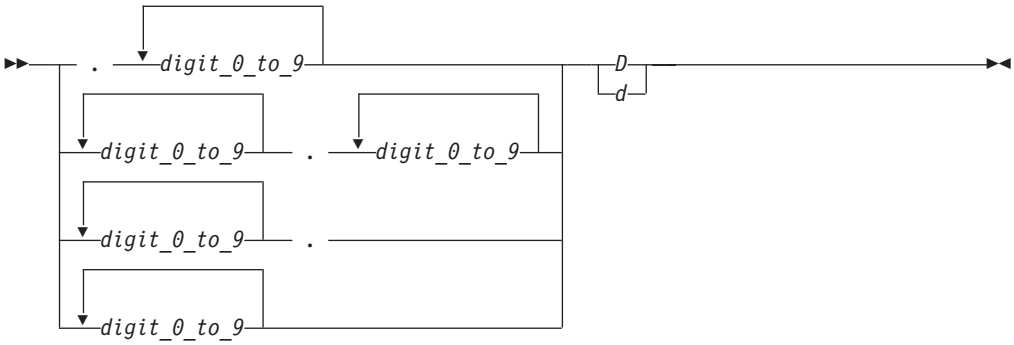
Complex floating-point types

Fixed-point decimal literals

C *Fixed-point decimal constants* are a z/OS XL C extension to Standard C. This type is available when you specify the LANGLVL(EXTENDED) compile-time option.

A fixed-point decimal constant has a numeric part and a suffix that specifies its type. The numeric part can include a digit sequence that represents the whole-number part, followed by a decimal point (.), followed by a digit sequence that represents the fraction part. Either the integral part or the fractional part, or both must be present.

A fixed-point constant has the form:



A fixed-point constant has two attributes:

- Number of digits (size)
- Number of decimal places (precision).

The suffix `D` or `d` indicates a fixed-point constant.

The following are examples of fixed-point decimal constants:

Fixed-point constant	(size, precision)
12345678901234567D	(16, 0)
12345678.12345678D	(16, 8)
12345678.d	(8, 0)
.1234567890d	(10, 10)
12345.99d	(7, 2)
000123.990d	(9, 3)
0.00D	(3, 2)

For more information on fixed-point decimal data types, see *z/OS XL C/C++ Programming Guide*. C

Related reference:

"Fixed point decimal types (C only)" on page 58

"The digitsof and precisionof operators (C only)" on page 160

Character literals

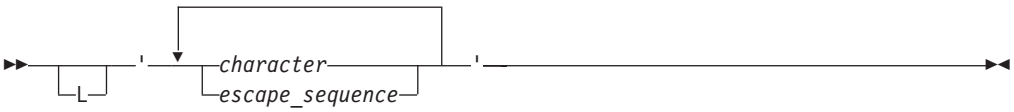
A *character literal* contains a sequence of characters or escape sequences enclosed in single quotation mark symbols, for example `'c'`. A character literal may be

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*.

C The type of a narrow character literal is `int`. The type of a wide character literal is `wchar_t`. The type of a multicharacter literal is `int`.

C++ The type of a character literal that contains only one character is `char`, which is an integral type. The type of a wide character literal is `wchar_t`. The type of a multicharacter literal is `int`.

Character literal syntax



At least one character or escape sequence must appear in the character literal, and the character literal must appear on a single logical source line.

The characters can be from the source program character set. You can represent the double quotation mark symbol by itself, but to represent the single quotation mark symbol, you must use the backslash symbol followed by a single quotation mark symbol (`\'` escape sequence). (See "Escape sequences" on page 34 for a list of other characters that are represented by escape characters.)

Outside of the basic source character set, the universal character names for letters and digits are allowed in C++ and at the C99 language level. C++ In C++, you must compile with the LANGLVL(UCS) option for universal character name support.

The following are examples of character literals:

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

Related reference:

"Character types" on page 59

"Source program character set" on page 32

"The Unicode standard" on page 35

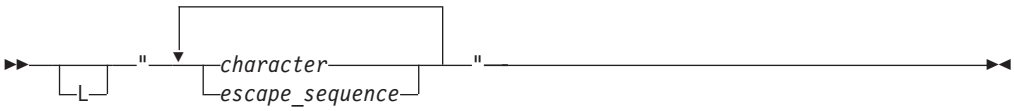
String literals

A *string literal* contains a sequence of characters or escape sequences enclosed in double quotation mark symbols. 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 narrow string literal is array of `char`. The type of a wide character string literal is array of `wchar_t` Both types have static storage duration.

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

String literal syntax



Multiple spaces contained within a string literal are retained.

Use the escape sequence `\n` to represent a new-line character as part of the string. Use the escape sequence `\\` to represent a backslash character as part of the string. You can represent a single quotation mark symbol either by itself or with the escape sequence `\'`. You must use the escape sequence `\"` to represent a double quotation mark.

Outside of the basic source character set, the universal character names for letters and digits are allowed in C++ and at the C99 language level. C++ In C++, you must compile with the **LANGLVL(UCS)** option for universal character name support.

See the following examples of string literals:

```
char titles[ ] = "Handel's \"Water Music\"";
char *temp_string = "abc" "def" "ghi";    // *temp_string = "abcdefghi\0"
wchar_t *wide_string = L"longstring";
```

This example illustrates escape sequences in string literals:

CCNX02K

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

int main () {
       char *s ="Hi there! \n";
      cout << s;
       char *p = "The backslash character \\.";
       cout << p << endl;
       char *q = "The double quotation mark \".\n";
       cout << q ;
 }
```

This program produces the following output:

```
Hi there! The backslash character \. The double quotation mark ".
```

To continue a string on the next line, use the line continuation character (\ symbol) followed by optional whitespace and a new-line character (required). For example:

```
char *mail_addr = "Last Name    First Name    MI   Street Address \
       893    City     Province   Postal code ";
```

Note: When a string literal appears more than once in the program source, how that string is stored depends on whether strings are read-only or writable. By default, the compiler considers strings to be read-only. z/OS XL C/C++ might allocate only one location for a read-only string; all occurrences refer to that one location. However, that area of storage is potentially write-protected. If strings are writable, then each occurrence of the string has a separate, distinct storage location

that is always modifiable. You can use the directive or the ROSTRING compiler option to change the default storage for string literals.

String concatenation

Another way to continue a string is to have two or more consecutive strings. Adjacent string literals can be concatenated to produce a single string. For example:

```
"hello " "there"    //equivalent to "hello there"
"hello" "there"     //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` .

If a wide string literal and a narrow string literal are adjacent, as in the following example:

```
"hello " L"there"
```

the result is a wide string literal.

Note: C In C99, narrow strings can be concatenated with wide string literals. C++11 In C++11, the changes to string literal concatenation in the C99 preprocessor are adopted to provide a common preprocessor interface for C and C++ compilers. Narrow strings can be concatenated with wide string literals in C++11. For more information, see “C99 preprocessor features adopted in C++11” on page 526.

Following any concatenation, `'\0'` of type `char` is appended at the end of each string. For a wide string literal, `'\0'` of type `wchar_t` is appended. By convention, programs recognize the end of a string by finding the null character. 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"
```

Related reference:

“Character types” on page 59

“Source program character set” on page 32

“The Unicode standard” on page 35

String concatenation of u-literals

Pointer literal (C++11)

The only pointer literal is the `nullptr` keyword that is a `prvalue` of type `std::nullptr_t`. A `prvalue` of this type is a null pointer constant that can be converted to any pointer type, pointer-to-member type, or bool type.

Related reference:

“Pointer conversions” on page 137

Punctuators and operators

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.

C99 and C++ define the following tokens as punctuators, operators, or preprocessing tokens:

Table 13. C and C++ punctuators

[ ]	( )	{ }	,	:	;
*	=	...	#		
.	->	++	--	##	
&	+	-	~	!	
/	%	<<	>>	!=	
<	>	<=	>=	==	
^	|	&&	||	?	
*=	/=	%=	+=	-=	
<<=	>>=	&=	^=	|=	

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

Table 14. C++ punctuators

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

C++

Alternative tokens

Both C and C++ provide the following alternative representations for some operators and punctuators. The alternative representations are also known as *digraphs*.

Operator or punctuator	Alternative representation
{	<%
}	%>
[	<:
]	:>
#	%:
##	%:%:

Note: The recognition of these alternative representations is controlled by the DIGRAPHS option; for more information, see "Digraph characters" on page 37.

In addition to the operators and punctuators listed above, C++ and C at the C99 language level provide the following alternative representations. In C, they are defined as macros in the header file `iso646.h`.

Operator or punctuator	Alternative representation
&&	and
|	bitor
||	or
^	xor
~	compl

Operator or punctuator	Alternative representation
&	bitand
&=	and_eq
|=	or_eq
^=	xor_eq
!	not
!=	not_eq

Related reference:

"Digraph characters" on page 37
"Boolean types" on page 55
"Boolean conversions" on page 130
"Floating-point types" on page 56
"Floating-point conversions" on page 130
"Unary expressions" on page 151
"Fixed point decimal types (C only)" on page 58
"The digitsof and precisionof operators (C only)" on page 160
"Source program character set"
"Character types" on page 59
Chapter 6, "Expressions and operators," on page 141

Source program character set

See the following list of the basic source character sets that are available at both compile time 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 1 2 3 4 5 6 7 8 9
- The following graphic characters:
  ! " # % & ' ( ) * + , - . / : ; < = > ? [\ ] _ { } ~
  - The caret (^) character in ASCII (bitwise exclusive OR symbol) or the equivalent not (¬) character in EBCDIC
  - The split vertical bar (¦) character in ASCII, which may be represented by the vertical bar (|) character on EBCDIC systems .
- The space character
- The control characters representing new-line, horizontal tab, vertical tab, form feed, end of string (NULL character), alert, backspace, and carriage return.

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

z/OS In a source file, a record contains one line of source text; the end of a record indicates the end of a source line.

If you use the **#pragma filetag** directive to specify the encoding of input files, the compiler converts this encoding to the encoding defined by code page IBM-1047. If you use the LOCALE to specify the encoding for output, the compiler converts the encoding from code page IBM-1047 to the encoding you have specified. These conversions apply to:

- Listings that contain identifier names and source code
- String literals and character constants that are emitted in the object code
- Messages generated by the compiler

They do not apply to source-code annotation in the pseudo-assembly listings.

Therefore, the encoding of the following characters from the basic character set may vary between the source-code generation environment and the runtime environment:

! # ' [ ] \ { } ~ ^ |

For a detailed description of the **#pragma filetag** directive and the LOCALE option, refer to the description of globalization, locales, and character sets in the *z/OS XL C/C++ User's Guide*. z/OS

Related reference:

Characters in identifiers

"#pragma filetag" on page 556

Multibyte characters

The compiler recognizes and supports the additional characters (the extended character set) which you can meaningfully use in string literals and character constants. The support for extended characters includes *multibyte character* sets. A *multibyte character* is a character whose bit representation fits into more than one byte.

z/OS systems represent multibyte characters by using Shiftout <SO> and Shiftin <SI> pairs. Strings are of the form:

```
<SO> x y z <SI>
```

Or they can be mixed:

```
<SO> x <SI> y z x <SO> y <SI> z
```

In the above, two bytes represent each character between the <SO> and <SI> pairs. z/OS XL C/C++ restricts multibyte characters to character constants, string constants, and comments.

Multibyte characters can appear in any of the following contexts:

- String literals and character constants. To declare a multibyte literal, use a wide-character representation, prefixed by `L`. For example:

  ```
  wchar_t *a = L"wide_char_string";
  wchar_t b = L'wide_char';
  ```

  Strings containing multibyte characters are treated essentially the same way as strings without multibyte characters. Generally, wide characters are permitted anywhere multibyte characters are, but they are incompatible with multibyte characters in the same string because their bit patterns differ. Wherever permitted, you can mix single-byte and multibyte characters in the same string.

- Preprocessor directives. The following preprocessor directives permit multibyte-character constants and string literals:
  - #define
  - #pragma comment
  - #include

  A file name specified in an `#include` directive can contain multibyte characters. For example:

  ```
  #include <multibyte_char/mydir/mysource/multibyte_char.h>
  #include "multibyte_char.h"
  ```
- Macro definitions. Because string literals and character constants can be part of `#define` statements, multibyte characters are also permitted in both object-like and function-like macro definitions.
- The # and ## operators.
- Program comments.

The following are restrictions on the use of multibyte characters:

- Multibyte characters are not permitted in identifiers.
- Hexadecimal values for multibyte characters must be in the range of the code page being used.
- You cannot mix wide characters and multibyte characters in macro definitions. For example, a macro expansion that concatenates a wide string and a multibyte string is not permitted.
- Assignment between wide characters and multibyte characters is not permitted.
- Concatenating wide character strings and multibyte character strings is not permitted.

Related reference:

Character literals

"The Unicode standard" on page 35

"Character types" on page 59

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 character syntax

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 wanted 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 *Information technology – Programming Languages – Universal Multiple-Octet Coded Character Set (UCS), ISO/IEC 10646:2003*. (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. According to the Unicode Standard, the encoding scheme of the standard is therefore sufficiently flexible to handle all known character encoding requirements, including coverage of all the world's historical scripts.

C99 allows the universal character name construct defined in ISO/IEC 10646 to represent characters outside the basic source character set. It permits universal character names in identifiers, character constants, and string literals. C++ To be compatible with C99, the z/OS XL C/C++ compiler supports universal character names as an IBM extension. In C++, you must compile with the LANGLVL(UCS) option for universal character name support. C++

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**
where N *is a hexadecimal digit*	
\UNNNNNNNN	NNNNNNNN
\uNNNN	0000NNNN

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.

UTF literals (IBM extension)

The ISO C and ISO C++ Committees have approved the implementation of *u-literals* and *U-literals* to support Unicode UTF-16 and UTF-32 character literals, respectively.

In C mode, the Unicode literals are enabled under the **EXTENDED** language level, and disabled under the strictly-conforming language levels. When the Unicode literals are enabled, the macro **__IBM_UTF_LITERAL** is predefined to 1, otherwise this macro is not predefined.

In C++ mode, to enable support for UTF literals in your source code, you must compile with the option **LANGLVL(EXTENDED0X)**. It can be customized through the option **[NO]KEYWORD(char16_t, char32_t)**. In C++ mode, the Unicode literals and character types are enabled under **EXTENDED** and **EXTENDED0X** language levels, and disabled under other language levels. When the Unicode literals are enabled, the macros **__IBM_UTF_LITERAL** and **__IBMCPP_UTF_LITERAL__** are predefined to 1, otherwise they are not predefined. Under the **EXTENDED** language level, the keywords **char16_t** and **char32_t** are disabled by default (but are available as typedefs via <**uchar.h**>). Under the **EXTENDED0X** language level, these keywords are enabled by default.

The following table shows the syntax for UTF literals.

Table 15. UTF literals

Syntax	Explanation
u'*character*'	Denotes a UTF-16 character.
u"*character-sequence*"	Denotes an array of UTF-16 characters.
U'*character*'	Denotes a UTF-32 character.
U"*character-sequence*"	Denotes an array of UTF-32 characters.

String concatenation of u-literals

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.

Related reference:

String concatenation

Digraph characters

You can represent unavailable characters in a source program by using a combination of two keystrokes that are called a *digraph character*. The preprocessor reads digraphs as tokens during the preprocessor phase. To enable processing of digraphs, use the DIGRAPH compiler option (which is enabled by default).

The digraph characters are:

%: or %%	#	number sign
<:	[	left bracket
:>	]	right bracket
<%	{	left brace
%>	}	right brace
%:%: or %%%%	##	preprocessor macro concatenation operator

You can create digraphs by using macro concatenation. z/OS XL C/C++ does not replace digraphs in string literals or in character literals. For example:

```
char *s = "<%%>; // stays "<%%>"

switch (c)  {
  case '<%' : { /* ... */ } // stays '<%'
  case '%>' : { /* ... */ } // stays '%>'
 }
```

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. For example,

```
some_array??(i??) = n;
```

Represents:

```
some_array[i] = n;
```

z/OS At compile time, the compiler translates the trigraphs found in string literals and character constants into the appropriate characters they represent. These characters are in the coded character set you select by using the LOCALE compiler option. If you do not specify the LOCALE option, the preprocessor uses code page IBM-1047.

The z/OS XL C/C++ compiler will compile source files that were edited using different encoding of character sets. However, they might not compile cleanly. z/OS XL C/C++ does not compile source files that you edit with the following:

- A character set that does not support all the characters that are specified above, even if the compiler can access those characters by a trigraph.
- A character set for which no one-to-one mapping exists between it and the character set above.

Note: The exclamation mark (!) is a variant character. Its recognition depends on whether or not the LOCALE option is active. For more information on variant characters, refer to the *z/OS XL C/C++ Programming Guide*. z/OS

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. C To enable C++ comments in C, you must compile with **c99**, or with the SSCOMM or LANGLVL(STDC99) or LANGLVL(EXTC99) options.

C

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 */.

You can also include multibyte characters.

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;
}
```

Note: You can also use the **#pragma comment** directive to place comments into an object module.

Related reference:

"#pragma comment" on page 541

"Multibyte characters" on page 33

Chapter 3. Data objects and declarations

The topics in this chapter discuss the various elements that constitute a declaration of a data object.

Topics are sequenced to loosely follow the order in which elements appear in a declaration. The discussion of the additional elements of data declarations is also continued in Chapter 4, “Declarators,” on page 97.

Overview of data objects and declarations

The following sections introduce some fundamental concepts regarding data objects and data declarations that will be used throughout this reference.

Overview of data objects

A data *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*. The data type of an object determines the storage allocation for that object and the interpretation of the values during subsequent access. It is also used in any type checking operations. Both the identifier and data type of an object are established in the object *declaration*.

C++ An instance of a class type is commonly called a *class object*. The individual class members are also called objects.

Data types are often grouped into type categories that overlap, such as:

Fundamental types versus derived types
Fundamental data types are also known as "basic", "fundamental" or "built-in" to the language. These include integers, floating-point numbers, and characters. *Derived* types, also known as "compound" types in Standard C++, are created from the set of basic types, and include arrays, pointers, structures, unions, enumerations. All C++ classes are considered compound types.

Built-in types versus user-defined types
Built-in data types include all of the fundamental types, plus types that refer to the addresses of basic types, such as arrays and pointers. *User-defined* types are created by the user from the set of basic types, in `typedef`, structure, union, and enumeration definitions. C++ classes are considered user-defined types.

Scalar types versus aggregate types
Scalar types represent a single data value, while *aggregate* types represent multiple values, of the same type or of different types. Scalars include the arithmetic types and pointers. Aggregate types include arrays, structures. C++ classes are considered aggregate types.

The following matrix lists the supported data types and their classification into fundamental, derived, scalar, and aggregate types.

Table 16. C/C++ data types

Data object	Basic	Compound	Built-in	User-defined	Scalar	Aggregate
integer types	+		+		+	
floating-point types[1]	+		+		+	
character types			+		+	
Booleans	+		+		+	
void type	+[2]		+		+	
pointers		+	+		+	
arrays		+	+			+
structures		+		+		+
unions		+		+		
enumerations		+		+	see note[3]	
C++ classes		+		+		+

Note:

1. C Although complex floating-point types are represented internally as an array of two elements, they behave in the same way as real floating-pointing types in terms of alignment and arithmetic operations, and can therefore be considered scalar types.
2. The `void` type is really an incomplete type, as discussed in "Incomplete types." Nevertheless, Standard C++ defines it as a fundamental type.
3. C The C standard does not classify enumerations as either scalar or aggregate. C++ Standard C++ classifies enumerations as scalars.

Incomplete types

The following are incomplete types:

- The `void` type
- Arrays 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

C However, if an 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. For more information, see "Variable length arrays" on page 106.

The following examples illustrate incomplete types:

```
void *incomplete_ptr;
struct dimension linear; /* no previous definition of dimension */
```

Compatible and composite types

C

In C, compatible types are defined as:

- two types that can be used together without modification (as in an assignment expression)
- two types that can be substituted one for the other without modification

A *composite type* is constructed from 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 identical are compatible; their composite type is the same type. Less obvious are the rules governing type compatibility of non-identical types, user-defined types, type-qualified types, and so on. "Type specifiers" on page 54 discusses compatibility for basic and user-defined types in C.

C

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.

Related reference:

Chapter 11, "Classes (C++ only)," on page 347

"The void type" on page 59

"Incomplete class declarations" on page 352

"Compatibility of arrays (C only)" on page 107

"Compatibility of pointers (C only)" on page 103

"Compatible functions (C only)" on page 224

Overview of data declarations and definitions

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

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. 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;`

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

Note: C The C99 standard no longer requires that all declarations appear at the beginning of a function before the first statement. C++ As in C++, you can mix declarations with other statements in your code.

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

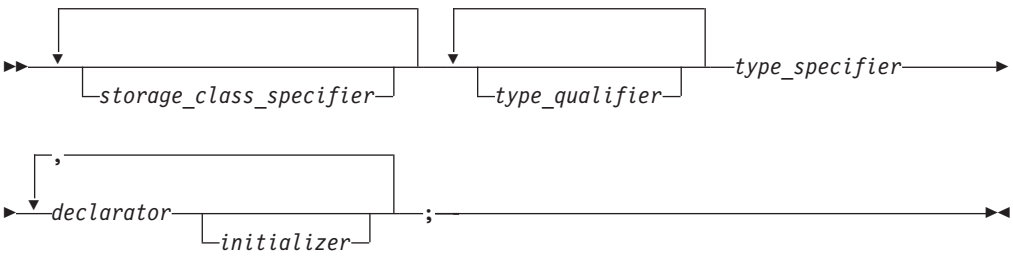
The elements of a declaration for a data object are as follows:

- "Storage class specifiers" on page 48, which specify storage duration and linkage
- "Type specifiers" on page 54, which specify data types
- "Type qualifiers" on page 85, which specify the mutability of data values
- Declarators, which introduce and include identifiers
- "Initializers" on page 108, which initialize storage with initial values

IBM In addition, for compatibility with GCC, z/OS XL C/C++ allows you to use *attributes* to modify the properties of data objects. They are described in "Variable attributes (IBM extension)" on page 125. IBM

All declarations have the form:

Data declaration syntax



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.

C 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 reference:

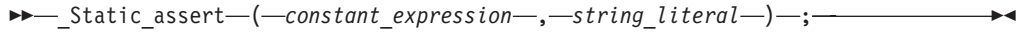
"Function declarations and definitions" on page 219

_Static_assert declaration (C11)

Note: IBM supports selected features of C11, known as C1X before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C11 features is complete, including the support of a new C11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the C11 features.

Static assertions can be declared to detect and diagnose common usage errors at compile time. A `_Static_assert` declaration takes the following form:

_Static_assert declaration syntax



The *constant_expression* must be an integer constant expression. If the integer constant expression evaluates to 0, the compiler issues a severe error containing the *string literal* with the source location of the `_Static_assert` declaration. Otherwise, the `_Static_assert` declaration has no effect.

The declaration of static assertions does not declare a new type or object, and does not imply any size or time cost at run time.

`static_assert` is a macro defined in `assert.h` for C.

The addition of static assertions to the C language has the following benefits:

- Libraries can detect common usage errors at compile time.
- Implementations of the C Standard Library can detect and diagnose common usage errors, improving usability.

You can declare static assertions to check important program invariants at compile time.

Examples: _Static_assert declaration

Example 1: The following example demonstrates the use of a `_Static_assert` declaration inside a structure.

```
#include <stddef.h>
struct __attribute__((packed)) B{
   char a;
   int i;
};

struct A{
   struct B b;
   _Static_assert(offsetof(struct B,i)==1,"S not packed");
};
```

Example 2: The following example contains static assertions declared with `static_assert`, so the `assert.h` header file must be included.

```
/* static_assert requires <assert.h> */
#include <assert.h>
static_assert(sizeof(long) >= 8, "64-bit not enabled.");
```

Example 3: The following example shows the use of a `_Static_assert` declaration with an invalid constant expression.

```
_Static_assert(1 / 0, "never shows up!");
```

When you compile this program, the compiler does not show the *string literal* in the `_Static_assert` declaration. Instead, the compiler issues an error message indicating that the divisor cannot be zero.

Related reference:

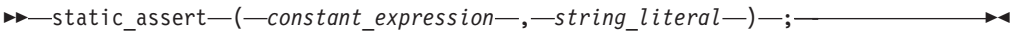
"Extensions for C11 compatibility" on page 640

static_assert declaration (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

Static assertions can be declared to detect and diagnose common usage errors at compile time. A `static_assert` declaration takes the following form:

static_assert declaration syntax



The *constant_expression* must be a constant expression that can be contextually converted to `bool`. If the value of the expression converted in such a way is false, the compiler issues a severe error containing the *string literal* with the source location of the `static_assert` declaration. Otherwise, the `static_assert` declaration has no effect.

You can declare static assertions anywhere that you use a `using` declaration, including namespace scope, block scope, and class member declaration lists.

The declaration of static assertions does not declare a new type or object, and does not imply any size or time cost at run time.

The C++ programming language also supports the `_Static_assert` keyword in all language levels for improved compatibility with the C programming language.

The addition of static assertions to the C++ language has the following benefits:

- Libraries can detect common usage errors at compile time.
- Implementations of the C++ Standard Library can detect and diagnose common usage errors, improving usability.

You can declare static assertions to check important program invariants at compile time.

Examples: static_assert declaration

The following example illustrates the use of a `static_assert` declaration in namespace scope.

```
static_assert(sizeof(long) >= 8, "64-bit code generation not
enabled/supported.");
```

The following example demonstrates the use of a `static_assert` declaration in class scope, with templates.

```
#include <type_traits>
#include <string>

template<typename T>
struct X {
   static_assert(std::tr1::is_pod<T>::value, "POD required to
instantiate class template X.");
   // ...
};

int main() {
   X<std::string> x;
}
```

The following example demonstrates the use of a `static_assert` declaration in block scope, with templates:

```
template <typename T, int N>
void f() {
   static_assert (N >=0, "length of array a is negative.");
   T a[N];
   // ...
}

int main() {
   f<int, -1>();
}
```

The following example shows the use of a `static_assert` declaration with an invalid constant expression.

```
static_assert(1 / 0, "never shows up!");
```

When you compile this program, the compiler does not show the *string literal* in the `static_assert` declaration. Instead, the compiler issues an error message indicating that the divisor cannot be zero.

Related reference:

"C++11 compatibility" on page 640

Storage class specifiers

A storage class specifier is used to refine the declaration of a variable, a function, and parameters. Storage classes determine 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 of 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 maintained throughout program run time or 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 use a storage class specifier to explicitly set its storage class.

C++11

In C++11, the keyword `auto` is no longer used as a storage class specifier. Instead, it is used as a type specifier. The compiler deduces the type of an `auto` variable from the type of its initializer expression. For more information, see "The auto type specifier (C++11)" on page 76.

The keyword `extern` was previously used as a storage specifier or as part of a linkage specification. The C++11 standard adds a third usage to use this keyword to specify explicit instantiation declarations. For more information, see "Explicit instantiation" on page 456.

C++11

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

- auto
- static
- extern
- C++ mutable
- register

Related reference:

"Function storage class specifiers" on page 225

"Initializers" on page 108

The 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.

Storage duration of automatic variables

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, a new object is allocated at each invocation of the block.

Linkage of automatic variables

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

Note: C++11 In C++11, the keyword `auto` is no longer used as a storage class specifier. Instead, it is used as a type specifier. The compiler deduces the type of an `auto` variable from the type of its initializer expression. For more information, see "The auto type specifier (C++11)" on page 76.

Related reference:

"Initialization and storage classes" on page 109

"Block statements" on page 199

"The goto statement" on page 213

The static storage class specifier

Objects declared with the `static` storage class specifier 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 a variable is different from file or global scope: a variable can have static duration but local scope.

C The keyword `static` is the major mechanism in C to enforce information hiding.

C++ C++ enforces information hiding through the namespace language feature and the access control of classes. The use of the keyword `static` to limit the scope of external variables is deprecated for declaring objects in namespace scope.

The `static` storage class specifier can be applied to the following declarations:

- Data objects
- C++ Class members

- Anonymous unions

You cannot use the `static` storage class specifier with the following:

- Type declarations
- Function parameters

C 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. See "Static array indices in function parameter declarations (C only)" on page 238 for more information.

Linkage of static variables

If a declaration of an object contains the `static` storage class specifier and has file scope, the identifier has internal linkage. Each instance of the particular identifier therefore represents the same object within one file only. If a declaration of an object contains the static storage class specifier and has function scope, an object is statically allocated and all the function calls use the same object. For example, if a static variable `x` has been declared in function `f`, when the program exits the scope of `f`, `x` is not destroyed:

```
#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 function local static variable, it is not reinitialized to `0` on successive calls to `f`.

Related reference:

"The static storage class specifier" on page 226

"Static members" on page 366

"Initialization and storage classes" on page 109

"Internal linkage" on page 7

Chapter 9, "Namespaces (C++ only)," on page 317

The extern storage class specifier

The extern storage class specifier lets you declare objects that several source files can use. An extern declaration makes the described variable 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 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.

C++ 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. An extern declaration cannot appear in class scope.

Storage duration of external variables

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 of external variables

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.

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.

Note: C++11 The keyword extern was previously used as a storage specifier or as part of a linkage specification. The C++11 standard adds a third usage to use this keyword to specify explicit instantiation declarations. For more information, see "Explicit instantiation" on page 456.

Related reference:

"External linkage" on page 8

"Initialization and storage classes" on page 109

"The extern storage class specifier" on page 226

Chapter 9, "Namespaces (C++ only)," on page 317

"Class scope (C++ only)" on page 4

The mutable storage class specifier (C++ only)

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.

In the following example:

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

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

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`.

Related reference:

"Type qualifiers" on page 85

"References (C++ only)" on page 107

The register storage class specifier

The `register` storage class specifier indicates to the compiler that the object should be stored in a machine register. The `register` storage class specifier is typically specified for heavily used variables, such as a loop control variable, in the hopes of enhancing performance by minimizing access time. However, 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`.

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

The following restrictions apply to the `register` storage class specifier:

- C You cannot use pointers to reference objects that have the `register` storage class specifier.
- C You cannot use the `register` storage class specifier when declaring objects in global scope.
- C A register does not have an address. Therefore, you cannot apply the address operator (`&`) to a `register` variable.
- C++ You cannot use the `register` storage class specifier when declaring objects in namespace scope.

C++ 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
```

Storage duration of register variables

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, a new object is allocated at each invocation of the block.

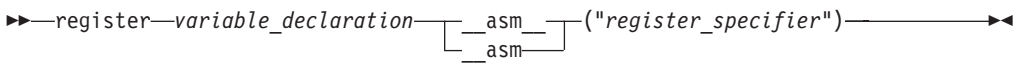
Linkage of register variables

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

Variables in specified registers (C only) (IBM extension)

When the GENASM compiler option is in effect, you can specify that a particular hardware register is dedicated to a global variable by using an `asm` *register variable* declaration. Global register variables reserve registers throughout the program; stores into the reserved register are never deleted. The register variable must be of type pointer.

Register variable declaration syntax



The *register_specifier* is a string representing a hardware register. The register name is CPU-specific. The following are valid register names:

r0 to r15 or R0 to R15
General purpose registers

The following are the rules of use for register variables:

- Registers can only be reserved for variables of pointer type.
- A global register variable cannot be initialized.
- The register dedicated for a global register variable should not be a volatile register, or the value stored into the global variable might not be preserved across a function call.
- More than one register variable can reserve the same register; however, the two variables become aliases of each other, and this is diagnosed with a warning.
- The same global register variable cannot reserve more than one register.

C++11

Note: The `register` storage class specifier is deprecated in C++11.

Related reference:

"Initialization and storage classes" on page 109

"Block/local scope" on page 2

"References (C++ only)" on page 107

"Inline assembly statements (IBM extension)" on page 215

Type specifiers

Type specifiers indicate the type of the object being declared. See the following available kinds of types:

- Fundamental or built-in types:
  - Arithmetic types
    - Integral types
    - Boolean types
    - Floating-point types
    - z/OS Fixed-point decimal types
    - Character types
  - The void type
- User-defined types

C++

A type is a literal type if it satisfies one of the following conditions:

- It is a scalar type.
- It is a reference type.
- It is an array of literal type.
- C++11 It is a class type with all the following properties:
  - The class has a trivial destructor.
  - Each constructor call and full expression in the initializers for nonstatic data members (if any) is a constant expression.
  - The class is an aggregate type or has at least one `constexpr` constructor or constructor template that is not a copy or move constructor.
  - All nonstatic data members and base classes of the class are of literal types.

    C++11

C++

C++11

In the C++11 standard, the following type specifiers are introduced:

- The auto type specifier
- The decltype(*expression*) type specifier

C++11

Related reference:

“Function return type specifiers” on page 234

See "C/C++ data mapping" under "Implementation-defined behavior"

“Command-line arguments” on page 248

Integral types

Integer types fall into the following categories:

- 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

C++ z/OS XL C++ supports the `long long` data type for language levels other than ANSI by default. You can also control the support for `long long` using the LONGLONG suboption of LANGLVL. For example, specifying LANGLVL(ANSI, LONGLONG) would add the `long long` data type to the ISO language level. Refer to the *z/OS XL C/C++ User's Guide* for information on using the LANGLVL option.

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.

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.

Related reference:

Integer literals

"Integral conversions" on page 130

"Arithmetic conversions and promotions" on page 129

Chapter 10, "Overloading (C++ only)," on page 327

See "Integers" under "Implementation-defined behavior"

Boolean types

A Boolean variable can be used to hold the integer values `0` or `1`, C++ or the literals `true` or `false` C++, which are implicitly promoted to the integers `1` and `0` respectively, whenever an arithmetic value is necessary. The Boolean type is unsigned and has the lowest ranking in its category of standard unsigned integer types; it may not be further qualified by the specifiers `signed`, `unsigned`, `short`, or `long`. In simple assignments, if the left operand is a Boolean type, then the right operand must be either an arithmetic type or a pointer.

C Boolean type is a C99 feature. To declare a Boolean variable, use the `_Bool` type specifier. C

C++ To declare a Boolean variable in C++, use the `bool` type specifier. The result of the equality, relational, and logical operators is of type `bool`: either of the Boolean constants `true` or `false`. C++

You can use Boolean types 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`.

Related reference:

Boolean literals

"Boolean conversions" on page 130

Floating-point types

Floating-point type specifiers fall into the following categories:

- Real floating-point types
- Complex floating-point types

Real floating-point types

Generic, or binary, floating-point types consist of the following:

- float
- double
- long double

IBM Decimal floating-point types consist of the following:

- _Decimal32
- _Decimal64
- _Decimal128

Note: In order for the `_Decimal32`, `_Decimal64`, and `_Decimal128` keywords to be recognized, you must compile with the DFP compiler option. See DFP compiler option in the *z/OS XL C/C++ User's Guide* for details.

IBM

The magnitude ranges of the real floating-point types are given in the following table.

Table 17. Magnitude ranges of real floating-point types

Type	Range
FLOAT(HEX):	
float	5.397605^{-79} - 7.237005^{75}
double	5.397605^{-79} - 7.237006^{75}
long double	5.397605^{-79} - 7.237006^{75}
FLOAT(IEEE):	
float	1.175494^{-38} - 3.402823^{38}
double	2.225074^{-308} - 1.797693^{308}
long double	3.362103^{-4932} - 1.189731^{4932}
DFP:	
_Decimal32	0.000001^{-95} to 9.999999^{96}

Table 17. Magnitude ranges of real floating-point types (continued)

Type	Range
_Decimal64	0.000000000000001^{-383} to 9.999999999999999^{384}
_Decimal128	$0.00000000000000000000000000000001^{-6143}$ to $9.999999999999999999999999999999999^{6144}$

If a floating-point constant is too large or too small, the result is undefined by the language.

z/OS Note that z/OS XL C/C++ supports IEEE binary floating-point variables as well as IBM z/Architecture® hexadecimal floating-point variables. For details on the FLOAT compiler option, see the *z/OS XL C/C++ User's Guide*.

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.

IBM You can use decimal floating-point types with any of the operators that are supported for binary floating-point types. You can also perform implicit or explicit conversions between decimal floating-point types and all other integral types, generic floating-point types, or packed decimals. However, there are restrictions on the use of decimal floating-point types with other arithmetic types as follows:

- You cannot mix decimal floating-point types with generic floating-point types or complex floating-point types in arithmetic expressions, unless you use explicit conversions.
- Implicit conversion between decimal floating-point types and real binary floating-point types is only allowed via assignment, with the simple assignment operator =. Implicit conversion is performed in simple assignments, which also include function argument assignments and function return values. See "Floating-point conversions" on page 130 for details.

IBM

Complex floating-point types

Complex floating-point types are introduced in the C99 standard. C++ The z/OS XL C/C++ compiler supports this feature as an IBM extension. C++ The complex floating-point type specifiers are as follows:

- float _Complex
- double _Complex
- long double _Complex

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.

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.

IBM As an extension to C99, complex numbers may also be operands to the unary operators ++ (increment), -- (decrement), and ~ (bitwise negation). IBM

Related reference:

Floating-point literals

“Floating-point conversions” on page 130

“Arithmetic conversions and promotions” on page 129

See "Floating-point numbers" under "Implementation-defined behavior"

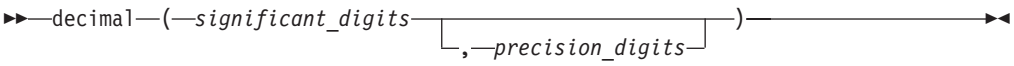
Complex literals (C only)

“The __real__ and __imag__ operators (IBM extension)” on page 160

Fixed point decimal types (C only)

Fixed point decimal types are classified as arithmetic types. To declare fixed point decimal variables and initialize them with fixed point decimal constants, you use the type specifier `decimal`. For this type specifier, `decimal` is a macro that is defined in the `decimal.h` header file. Remember to include `decimal.h` if you use fixed point decimals in your program.

Fixed point decimal syntax



The *significant_digits* is a positive integral constant expression. The second argument, *precision_digits* is optional. If you leave it out, the default value is 0. The type specifiers `decimal(`*n*`,0)` and `decimal(`*n*`)` are type-compatible.

In the type specifier, *significant_digits* and *precision_digits* have a range of allowed values according to the following rules:

1. *precision_digits* <= *significant_digits*
2. 1 <= *significant_digits* <= `DEC_DIG`
3. 0 <= *precision_digits* <= `DEC_PRECISION`

The decimal.h file defines DEC_DIG (the maximum number of digits) and DEC_PRECISION (the maximum precision). Currently, it uses a maximum of 31 digits for both limits.

The following examples show how to declare a variable as a fixed point decimal data type:

```
decimal(10,2)  x;
decimal(5,0)   y;
decimal(5)     z;
decimal(18,10) *ptr;
decimal(8,2)   arr[100];
```

In the previous example:

- `x` can have values between -99999999.99D and +99999999.99D.
- `y` and `z` can have values between -99999D and +99999D.
- `ptr` is a pointer to type `decimal(18,10)`.
- `arr` is an array of 100 elements, where each element is of type `decimal(8,2)`.

Related reference:

Fixed-point decimal literals (z/OS only)

“The digitsof and precisionof operators (C only)” on page 160

Character types

Character types fall into the following categories:

- Narrow character types:
  - char
  - signed char
  - unsigned char
- Wide character type `wchar_t`

The `char` specifier is an integral type. 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'`)

C A `char` is a distinct type from `signed char` and `unsigned char`, and the three types are not compatible.

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

If it does not matter if a `char` data object is `signed` or `unsigned`, you can declare the object as having the data type `char`. Otherwise, explicitly declare `signed char` or `unsigned char` to declare numeric variables that occupy a single byte. When a `char` (`signed` or `unsigned`) is widened to an `int`, its value is preserved.

By default, `char` behaves like an `unsigned char`. To change this default, you can use the CHARS option or the **#pragma chars** directive. See "#pragma chars" on page 539 and CHARS in the *z/OS XL C/C++ User's Guide* for more information.

Related reference:

Character literals

String literals

"Arithmetic conversions and promotions" on page 129

The 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.

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 cases:

- An expression statement
- The left operand of a comma expression
- The second or third operand in a conditional expression.

Related reference:

"Pointers" on page 100

"Comma operator ," on page 172

"Conditional expressions" on page 174

"Function declarations and definitions" on page 219

The atomic type (C11)

Atomic declaration syntax

The *type_name* cannot be an array type, a function type, an atomic type, or a qualified type.

The properties associated with atomic types are meaningful only for expressions that are lvalues. If the `_Atomic` keyword is immediately followed by a left parenthesis, it is used a type specifier rather than a type qualifier.

The following example shows the declaration of atomic objects:

```
_Atomic(float) a;
_Atomic(int) b;
_Atomic int c;
```

To support the compatibility of atomic types between C and C++, you can define a macro `#define _Atomic(T) atomic<T>`.

User-defined types

See the following user-defined types:

- Structures and unions
- Enumerations
- Typedef definitions
- C++ Classes
- C++ Elaborated type specifiers

C++ classes are discussed in Chapter 11, "Classes (C++ only)," on page 347. Elaborated type specifiers are discussed in "Scope of class names" on page 351.

Structures and unions

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*.

A *union* is an object similar to a structure except that all of its members start at the same location in memory. A union variable can represent the value of only one of its members at a time.

C++ In C++, structures and unions are the same as classes except that their members and inheritance are public by default. C++

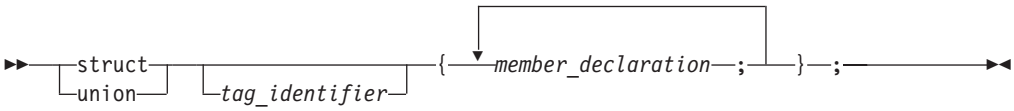
You can declare a structure or union type separately from the definition of variables of that type, as described in "Structure and union type definition" on page 61 and "Structure and union variable declarations" on page 66; or you can define a structure or union data type and all variables that have that type in one statement, as described in "Structure and union type and variable definitions in a single statement" on page 67.

Structures and unions are subject to alignment considerations. For information about changing alignment and packing structures, see “The _Packed qualifier (C only)” on page 124 and “#pragma pack” on page 583.

Structure and union type definition

A structure or union *type definition* contains the `struct` or `union` keyword followed by an optional identifier (the structure tag) and a brace-enclosed list of members.

Structure or union type definition syntax



The *tag_identifier* gives a name to the type. If you do not provide a tag name, you must put all variable definitions that refer to the type within the declaration of the type, as described in “Structure and union type and variable definitions in a single statement” on page 67. Similarly, you cannot use a type qualifier with a structure or union definition; type qualifiers placed in front of the `struct` or `union` keyword can only apply to variables that are declared within the type definition.

Member declarations

The list of members provides a structure or union data type with a description of the values that can be stored in the structure or union. The definition of a member has the form of a standard variable declaration. The names of member variables must be distinct within a single structure or union, but the same member name may be used in another structure or union type that is defined within the same scope, and may even be the same as a variable, function, or type name.

A structure or union member may be of any type except:

- any variably modified type
- `void` type
- C a function
- any incomplete type

Because incomplete types are not allowed as members, a structure or union type may not contain an instance of itself as a member, but is allowed to contain a pointer to an instance of itself. As a special case, the last member of a structure with more than one member may have an incomplete array type, which is called a *flexible array member*, as described in Flexible array members.

IBM

As an extension to Standard C and C++ for compatibility with GNU C/C++, z/OS XL C/C++ also allows zero-extent arrays as members of structures and unions, as described in Zero-extent array members (IBM extension). IBM

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`. C++

A member that does not represent a bit field can be qualified with either of the type qualifiers `volatile` or `const`. The result is an lvalue.

Structure members are assigned to memory addresses in increasing order, with the first component starting at the beginning address of the structure name itself. To allow proper alignment of components, padding bytes may appear between any consecutive members in the structure layout.

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). 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.

Flexible array members

A *flexible array member* is an unbounded array that occurs within a structure. It is a C99 feature and C++ the z/OS XL C/C++ compiler supports it as an IBM extension C++ . Flexible array members can be used to access a variable-length object. A flexible array member is permitted as the last member of a structure, provided that the structure has more than one named member. It is declared with an empty index as follows:

array_identifier [ ];

For example, `b` is a flexible array member of structure `f`.

```
struct f{
  int a;
  int b[];
};
```

Because a flexible array member has an incomplete type, you cannot apply the `sizeof` operator to a flexible array. In this example, the statement `sizeof(f)` returns the same result as `sizeof(f.a)`, which is the size of an integer. The statement `sizeof(f.b)` cannot be used, because `b` is a flexible array member that has an incomplete type.

Any structure containing a flexible array member cannot be a member of another structure or an element of an array, for example:

```
struct f{
  int a;
  int b[];
};
struct f fa[10]; // Error.
```

IBM To be compatible with GNU C/C++, the z/OS XL C/C++compiler extends Standard C and C++, to ease the restrictions on flexible array members and allow the following situations:

- Flexible array members can be declared in any part of a structure, not just as the last member. The type of any member that follows the flexible array member is not required to be compatible with the type of the flexible array member; however, a warning message is issued when a flexible array member is followed by members of an incompatible type. The following example demonstrates this:

```
struct s {
  int a;
  int b[];
  char c; // The compiler issues a warning message.
} f;
```

- Structures containing flexible array members can be members of other structures.
- C Flexible array members can be statically initialized only if either of the following two conditions is true:
  - The flexible array member is the last member of the structure, for example:

```
struct f {
  int a;
  int b[];
} f1 = {1,{1,2,3}}; // Fine.

struct a {
  int b;
  int c[];
  int d[];
} e = { 1,{1,2},3}; // Error, c is not the last member
                    // of structure a.
```

  - Flexible array members are contained in the outermost structure of nested structures. Members of inner structures cannot be statically initialized, for example:

```
struct b {
  int c;
  int d[];
};

struct c {
  struct b f;
  int g[];
} h ={{1,{1,2}},{1,2}}; // Error, member d of structure b is
                        // in the inner nested structure.
```

C

IBM

Zero-extent array members (IBM extension)

Zero-extent arrays are provided for GNU C/C++ compatibility, and can be used to access a variable-length object.

A zero-extent array is an array with an explicit zero specified as its dimension.

array_identifier [0]

For example, b is a zero-extent array member of structure f.

```
struct f{
  int a;
  int b[0];
};
```

The sizeof operator can be applied to a zero-extent array, and the value returned is 0. In this example, the statement sizeof(f) returns the same result as sizeof(f.a), which is the size of an integer. The statement sizeof(f.b) returns 0.

A structure containing a zero-extent array can be an element of an array, for example:

```
struct f{
  int a;
  int b[0];
};
struct f fa[10]; // Fine.
```

A zero-extent array can only be statically initialized with an empty set {}. Otherwise, it must be initialized as a dynamically allocated array. For example:

```
struct f{
  int a;
  int b[0];
};
struct f f1 = {100, {}}; //Fine.
struct f f2 = {100, {1, 2}}; //Error.
```

If a zero-extent array is not initialized, no static zero filling occurs, because a zero-extent array is defined to have no members. The following example demonstrates this:

```
#include <stdio.h>

struct s {
  int a;
  int b[0];
};

struct t1 {
  struct s f;
  int c[3];
} g1 = {{1},{1,2}};

struct t2 {
  struct s f;
  int c[3];
} g2 = {{1,{}},{1,2}};

int main() {
  printf("%d %d %d %d\n", g1.f.a, g1.f.b[0], g1.f.b[1], g1.f.b[2]);
  printf("%d %d %d %d\n", g2.f.a, g2.f.b[0], g2.f.b[1], g2.f.b[2]);
  return 0;
}
```

In this example, the two `printf` statements produce the same output:

```
1 1 2 0
```

A zero-extent array can be declared in any part of a structure, not just as the last member. The type of any member following the zero-extent array is not required to be compatible with the type of the zero-extent array; however, a warning is issued when a zero-extent array is followed by members of an incompatible type. For example:

```
struct s {
  int a;
  int b[0];
  char c;    // Issues a warning message
} f;
```

You can declare a zero extent array only as a member of an aggregate type. For example:

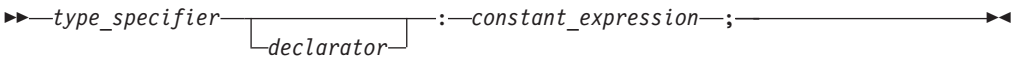
```
int func(){
  int a[0];        // error
  struct S{
```

```
    int x;
    char b[0];     // fine
  };
}
```

Bit field members

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.

Bit field member declaration syntax



The *constant_expression* is a constant integer expression that indicates the field width in bits. A bit field declaration may not use either of the type qualifiers `const` or `volatile`.

C

In C99, the allowable data types for a bit field include `_Bool`, `int`, `signed int`, and `unsigned int`.

C11 The width of a `_Bool` bit field cannot be greater than one bit. C11

C

C++ A bit field can be any integral type or enumeration type. C++

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;
};
```

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.

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

Bit fields are bit packed. They can cross word and byte boundaries. No padding is inserted between two (non-zero length) bit field members. Bit padding can occur after a bit field member if the next member is a zero length bitfield or a non-bit field. Non-bit field members are aligned based on their declared type. For example,

the following structure demonstrates the lack of padding between bit field members, and the insertion of padding after a bit field member that precedes a non-bit field member.

```
struct {
  int larry : 25; // Bit Field: offset 0 bytes and 0 bits.
  int curly : 25; // Bit Field: offset 3 bytes and 1 bit (25 bits).
  int moe;        // non-Bit Field: offset 8 bytes and 0 bits (64 bits).
} stooges;
```

There is no padding between **larry** and **curly**. The bit offset of **curly** would be 25 bits. The member **moe** would be aligned on the next 4 byte boundary, causing 14 bits a padding between **curly** and **moe**.

Bit fields with a length of 0 must be unnamed. Unnamed bit fields cannot be referenced or initialized.

A zero-width bit field causes the next field to be aligned on the next container boundary. However, a `_Packed` (C only) structure, which has a zero-width bit field, causes the next field to be aligned on the next byte boundary.

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;
```

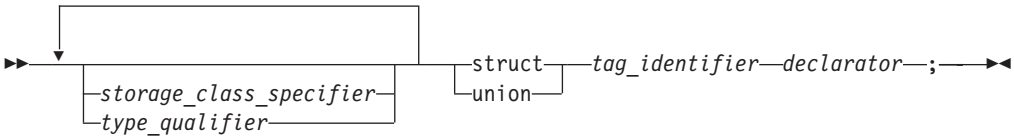
The structure `kitchen` contains eight members totalling 16 bytes. The following table describes the storage that each member occupies:

Member name	Storage occupied
`light`	1 bit
`toaster`	1 bit
(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

Structure and union variable declarations

A structure or union *declaration* has the same form as a definition except the declaration does not have a brace-enclosed list of members. You must declare the structure or union data type before you can define a variable having that type.

Structure or union variable declaration syntax



The *tag_identifier* indicates the data type of the structure or union.

C++ The keyword `struct` is optional in structure variable declarations. C++

You can declare structures or unions having any storage class. The storage class specifier and any type qualifiers for the variable must appear at the beginning of the statement. Structures or unions declared with the `register` storage class specifier are treated as automatic variables.

The following example defines structure type `address`:

```
struct address {
  int street_no;
  char *street_name;
  char *city;
  char *prov;
  char *postal_code;
};
```

The following examples declare two structure variables of type `address`:

```
struct address perm_address;
struct address temp_address;
```

Structure and union type and variable definitions in a single statement

You can define a structure (or union) type and a structure (or union) variable in one statement, by putting a declarator and an optional initializer after the variable definition. The following example defines a union data type (not named) and a union variable (named `length`):

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

Note that because this example does not name the data type, `length` is the only variable that can have this data type. Putting an identifier after `struct` or `union` keyword provides a name for the data type and lets you declare additional variables of this data type later in the program.

To specify a storage class specifier for the variable or variables, 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;
```

In this case, both perm_address and temp_address are assigned static storage.

Type qualifiers can be applied to the variable or variables declared in a type definition. Both of the following examples are valid:

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

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

In both cases, the structures file1 and file2 are qualified as volatile.

Access to structure and union members

Once structure or union variables have been declared, members are referenced by specifying the 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 the string "Ontario" to the pointer prov that is in the structure perm_address.

All references to members of structures and unions, including bit fields, must be fully qualified. In the previous example, the fourth field cannot be referenced by prov alone, but only by perm_address.prov.

Anonymous structures (C11)

Note: IBM supports selected features of C11, known as C1X before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C11 features is complete, including the support of a new C11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the C11 features.

An *anonymous structure* is a structure that does not have a tag or a name and that is a member of another structure or union. All the members of the anonymous structure behave as if they were members of the parent structure. An anonymous structure must meet the following conditions:

- The structure is nested inside another structure or union.
- The structure has no tag.
- The structure has no name.

For example, the following code fragment demonstrates the conditions that an anonymous structure must meet.

```
struct v {
  union {
     // This is an anonymous structure, because it has no tag, no name,
     // and is a member of another structure or union.
```

```
    struct { int i, j; };

    // This is not an anonymous structure, because it has a name.
    struct { long k, l; } w;

   // This is not an anonymous structure, because
   // the structure has a tag "phone".
   struct phone {int number, areanumber;};
  };

  int m;
} v1;
```

Anonymous unions

An *anonymous union* is a union that does not have a tag or a name and that is a member of another union or structure. It cannot be followed by a declarator. An anonymous union is not a type; it defines an unnamed object.

z/OS XL C supports anonymous unions only under extended language levels.

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, and it cannot have member functions. A global or namespace anonymous union must be declared with the keyword `static`. C++

C11

Related reference:

"Classes and structures" on page 350
"Variable length arrays" on page 106
"The aligned variable attribute" on page 126
"Initialization of structures and unions" on page 112
"Compatibility of structures, unions, and enumerations (C only)" on page 73
"Dot operator ." on page 150
"Arrow operator ->" on page 150
"Storage class specifiers" on page 48
"Type qualifiers" on page 85
"The static storage class specifier" on page 49
"Member functions" on page 359

Enumerations

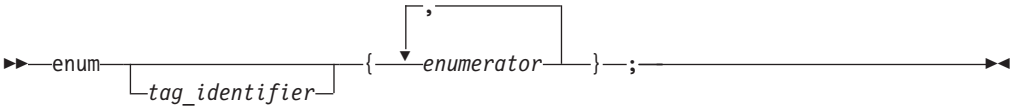
An *enumeration* is a data type that consists of a set of named values that represent integral constants, known as *enumeration constants*. An enumeration is also referred to as an *enumerated type* because you must list (enumerate) each of the values in creating a name for each of them. In addition 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 declare an enumeration type separately from the definition of variables of that type, as described in "Enumeration type definition" and "Enumeration variable declarations" on page 72; or you can define an enumeration data type and all variables that have that type in one statement, as described in "Enumeration type and variable definitions in a single statement" on page 72.

Enumeration type definition

An enumeration type definition contains the `enum` keyword followed by an optional identifier (the enumeration tag) and a brace-enclosed list of enumerators. A comma separates each enumerator in the enumerator list. C C99 allows a trailing comma between the last enumerator and the closing brace. C

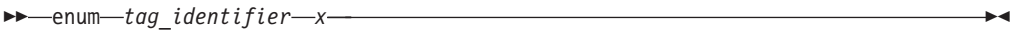
Enumeration definition syntax



The *tag_identifier* gives a name to the enumeration type. If you do not provide a tag name, you must put all variable definitions that refer to the enumeration type within the declaration of the type, as described in "Enumeration type and variable definitions in a single statement" on page 72. Similarly, you cannot use a type qualifier with an enumeration definition; type qualifiers placed in front of the `enum` keyword can only apply to variables that are declared within the type definition.

Elaborated type specifier

Elaborated type specifier syntax



The elaborated type specifier refers to a previously declared enumeration. The *x* is a variable that has the type *tag_identifier*.

The `enum` keyword can be used to refer to scoped or unscoped enumerations during variable declaration or definition. For example:

```
// a scoped enumeration
enum class color { red, white, black, yellow };

// an unscoped enumeration
enum letter {A, B, C, D};

// valid, regular type name usage
```

```
color pic1 = color :: white;

// valid, elaborated type usage
enum color pic2 = color :: red;
```

You cannot use `enum class` or `enum struct` in the elaborated type specifier. For example:

```
enum class color pic3 = color :: black;    // invalid
```

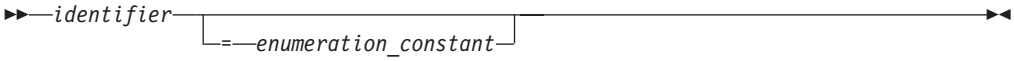
The elaborated type specifier for an unscoped enumeration is the same as that for a scoped enumeration. For example:

```
enum letter let1 = letter :: A;            // valid
```

Enumeration members

The list of enumeration members, or *enumerators*, provides the data type with a set of values.

Enumeration member declaration syntax



C In C, an *enumeration constant* is of 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`). Otherwise, the condition is tolerated, a diagnostic message is issued, but the value of the enumeration constant is undefined. C

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. You can use an enumeration constant anywhere an integer constant is allowed, or anywhere a value of the enumeration type is allowed. C++

The value of an enumeration 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 enumeration constant. The enumeration constant represents the value of the constant expression.
2. If no explicit value is assigned to the first enumerator, then it takes the value 0 (zero).
3. Enumeration constants with no explicitly assigned values receive the integer value that is one greater than the value represented by the previous enumeration constant.

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       */
```

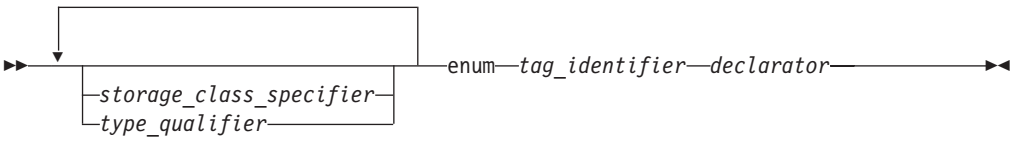
Each enumeration constant must be unique within the scope in which the enumeration is defined. In the following example, the second declarations of `average` and `poor` cause compiler errors:

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

Enumeration variable declarations

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

Enumeration variable declaration syntax



The *tag_identifier* indicates the previously-defined data type of the enumeration.

C++ The keyword `enum` is optional in enumeration variable declarations. C++

Enumeration type and variable definitions in a single statement

You can define a type and a variable in one statement by using a declarator and an optional initializer after the variable 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;
```

C++

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. However, you cannot declare any additional enumeration variables using this set of enumeration constants.

Related reference:

"Arithmetic conversions and promotions" on page 129

"#pragma enum" on page 549

"Integral types" on page 54

"Initialization of enumerations" on page 114

"Compatibility of structures, unions, and enumerations (C only)"

Compatibility of structures, unions, and enumerations (C only)

Within a single source file, each structure or union definition creates a new type that is neither the same as nor compatible with any other structure or union type. However, a type specifier that is a reference to a previously defined structure or union type is the same type. The 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 compatible in this example:

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

Compatible structures may be assigned to each other.

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

You cannot perform comparisons between packed and nonpacked structures or unions of the same type. You cannot assign packed and nonpacked structures or unions to each other, regardless of their type. You cannot pass a packed structure or union argument to a function that expects a nonpacked structure or union of the same type and vice versa.

Since the compiler treats enumeration variables and constants as integer types, you can freely mix the values of different enumerated types, regardless of type compatibility. Compatibility between an enumerated type and the integer type that represents it is controlled by compiler options and related pragmas. For a discussion of the ENUMSIZE compiler option, see the *z/OS XL C/C++ User's Guide*. For a discussion of the **#pragma enum** directive, see "#pragma enum" on page 549.

Compatibility across separate source files

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.

Related reference:

"Arithmetic conversions and promotions" on page 129

Chapter 11, "Classes (C++ only)," on page 347

Structure or union type definition

Incomplete types

"The _Packed qualifier (C only)" on page 124

"#pragma pack" on page 583

typedef definitions

Note: IBM supports selected features of C11, known as C1X before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C11 features is complete, including the support of a new C11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the C11 features.

You can use the`typedef` declaration to 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. 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.

C11

Using `typedef` redeclaration, you can redefine a name that is a previous `typedef` name in the same scope to refer to the same type if the type is not a variably modified type. For example:

```
typedef char AChar;
typedef char AChar;
```

IBM When any extended language level is in effect, typedef redeclaration supports all types, including a variably modified type. IBM

For more information about variably modified types, see "Variable length arrays" on page 106.

C11

Examples of typedef definitions

The following statements define 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 preceding declarations are equivalent to the following declaration:

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

Similarly, typedef can be used to define a 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 parameters, returning int".

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

In the following typedef definitions, 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++

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.

A C++ class defined in a typedef definition without being named is given a dummy name. Such a class cannot have constructors or destructors. Consider the following example:

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

In this example, an unnamed class is defined in a `typedef` definition. `Trees` is an alias for the unnamed class, but not the class type name. So you cannot define a destructor `~Trees()` for this unnamed class; otherwise, the compiler issues an error.

C++

C++11

Declaring typedef names as friends

In the C++11 standard, the extended friend declarations feature is introduced, with which you can declare `typedef` names as friends. For more information, see "Extended friend declarations" on page 374.

C++11

Related reference:

"Type names" on page 99

"Type specifiers" on page 54

"Structures and unions" on page 60

Chapter 11, "Classes (C++ only)," on page 347

"Friends" on page 373

The auto type specifier (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

C++11 introduces the keyword `auto` as a new type specifier. `auto` acts as a placeholder for a type to be deduced from the initializer expression of a variable. With auto type deduction enabled, you no longer need to specify a type while declaring a variable. Instead, the compiler deduces the type of an `auto` variable from the type of its initializer expression.

The following examples demonstrate the usage of auto type deduction.

```
auto x = 1;      //x : int

float* p;
auto x = p;      //x : float*
auto* y = p;     //y : float*

double f();
auto x = f();              //x : double
const auto& y = f();       //y : const double&

class R;
R* h();
auto* x = h();    //x : R*
auto y = h();     //y : R*
```

```
int& g();
auto x = g();           //x : int
const auto& y = g();    //y : const int&
auto* z = g();          //error, g() does not return a pointer type
```

By delegating the task of type deduction to the compiler, auto type deduction increases programming convenience, and potentially eliminates typing errors made by programmers. Auto type deduction also reduces the size and improves the readability of programs.

The following two examples demonstrate the benefits of enabling auto type deduction. The first example does not enable auto type deduction.

```
vector<int> vec;
for (vector<int>::iterator i = vec.begin(); i < vec.end(); i++)
{
   int* a = new int(1);
   //...
}
```

With auto type deduction enabled, the first example can be simplified as follows:

```
vector<int> vec;
for (auto i = vec.begin(); i < vec.end(); i++)
{
   auto a = new auto(1);
   //...
}
```

The following rules and constraints apply to the use of `auto` as a type specifier in auto type deduction.

- Auto type deduction cannot deduce array types.

  ```
  int x[5];
  auto y[5] = x;    //error, x decays to a pointer,
                    //which does not match the array type
  ```

- Auto type deduction cannot deduce *cv-qualifier* or reference type from the initializer.

  ```
  int f();
  auto& x = f();    //error, cannot bind a non-const reference
                    //to a temporary variable
  int& g();
  auto y = g();     //y is of type int
  auto& z = g();    //z is of type int&
  ```

- Auto type deduction supports multi-variable auto declarations. If the list of declarators contains more than one declarator, the type of each declarator can be deduced independently. If the deduced type is not the same in each deduction, the program is ill-formed.

  ```
  auto x=3, y=1.2, *z=new auto(1);     //error y: deduced as double,
                                       //but was previously deduced as int
  ```

- The name of the object that is declared can not be used in its initializer expression.

  ```
  auto x = x++;     //error
  ```

- `auto` can not be used in function parameters.

  ```
  int func(auto x = 3)     //error
  {
      //...
  }
  ```

Note: In C++11, the keyword `auto` is no longer used as a storage class specifier.

Related reference:

"Storage class specifiers" on page 48

"The auto storage class specifier" on page 49

"Type qualifiers" on page 85

"C++11 compatibility" on page 640

The decltype(*expression*) type specifier (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

The *decltype(expression)* specifier is a type specifier introduced in C++11. With this type specifier, you can get a type that is based on the resultant type of a possibly type-dependent expression.

decltype(*expression*) takes *expression* as an operand. When you define a variable by using decltype(*expression*), it can be thought of as being replaced by the compiler with the type or the derived type of *expression*. Consider the following example:

```
int i;
static const decltype(i) j = 4;
```

In this example, `decltype(i)` is equivalent to the type name `int`.

General rules for using decltype

When you use decltype(*expression*) to get a type, the following rules are applicable:

1. If *expression* is an unparenthesized *id-expression* or class member, decltype(*expression*) is the type of the entity named by *expression*. If there is no such entity, or if *expression* names a set of overloaded functions, the program is ill formed.
2. Otherwise, if *expression* is an xvalue, decltype(*expression*) is T&&, where T is the type of *expression*.
3. Otherwise, if *expression* is an lvalue, decltype(*expression*) is T&, where T is the type of *expression*.
4. Otherwise, decltype(*expression*) is the type of *expression*.

The following example illustrates how these rules are used:

```
const int* g(){
    return new int[0];
}

int&& fun(){
   int&& var = 1;
   return 1;
}

struct A{
    double x;
};
```

```
template <class T> T tf(const T& t){
    return t;
}

bool f(){
    return false;
}

struct str1{
    template <typename T, typename U>
    static decltype((*(T*)0) * (*(U*)0)) mult(const U& arg1, const T& arg2){
       return arg1 * arg2;
    }
};

template <typename T, typename U>  struct str2{
    typedef decltype((*(T*)0) + (*(U*)0)) btype;
    static btype g(T t, U u);
};

int main(){
    int i = 4;
    const int j = 6;
    const int& k = i;
    int&& m = 1;
    int a[5];
    int *p;

    decltype(i) var1;            // int
    decltype(1) var2;            // int
    decltype(2+3) var3;          // int(+ operator returns an rvalue)
    decltype(i=1) var4 = i;      // int&, because assignment to int
                                 // returns an lvalue
    decltype((i)) var5 = i;      // int&
    decltype(j) var6 = 1;        // const int
    decltype(k) var7 = j;        // const int&
    decltype("decltype") var8 = "decltype";     // const char(&)[9]
    decltype(a) var9;            // int[5]
    decltype(a[3]) var10 = i;    // int&([] returns an lvalue)
    decltype(*p)  var11 = i;     // int&(*operator returns an lvalue)
    decltype(fun()) var12 = 1;   // int&&
    decltype(tf(A())) var13;     // A
    decltype(f()) var14;         // bool
    decltype((f())) var15;       // bool, parentheses around f() are ignored
    decltype(f)  var16;          // bool()
    decltype(&f) var17;          // bool(*)()
    decltype(&A::x) var18;       // double A::*
    decltype(str1::mult(3.0, 4u))  var19;           // double
    decltype(str2<float, short>::g(1,3)) var20;     // float
    decltype(m) var21 = 1;       // int&&
    decltype((m)) var22 = m;     // int&
    return 0;
}
```

In this example, the comment after each decltype statement explains the type of the defined variable.

The following example illustrates an incorrect usage of decltype(*expression*):

```
int func(){
    return 0;
}
int func(int a){
    return 0;
}

int main(){
```

```
    int i = 4;

    // Incorrect usage. func names an overload function
    decltype(func) var1;

    // Correct usage. The overload operation is not ambiguous
    decltype(func(i)) var2;

    return 0;
}
```

In this example, the compiler issues an error message because it does not know which `func` function to match.

Rules for using decltype with structure member variables

When you use decltype(*expression*) to get a type, and *expression* is an unparenthesized member variable of an *object expression* (with a . operator) or a *pointer expression* (with a -> operator), the following rules apply:

- If the object expression or the pointer expression is specified with a constant or volatile qualifier, the type qualifier does not contribute to the result of decltype(*expression*).
- The lvalueness or rvalueness of the object expression or the pointer expression does not affect whether decltype(*expression*) is a reference type or not.

Example:

```
struct Foo{
    int x;
};

int main(){
    struct Foo f;
    const struct Foo g = {0};
    volatile struct Foo* h = &f;
    struct Foo func();

    decltype(g.x) var1;         // int
    decltype(h->x) var2;        // int
    decltype(func().x) var3;    // int
    return 0;
}
```

In this example, the constant qualifier of the object expression `g` is not desired in the result of `decltype(g.x)`. Similarly, the volatile qualifier of the pointer expression `h` is not desired in the result of `decltype(h->x)`. The object expression `g` and the pointer expression `h` are lvalues, and the object expression `func()` is an rvalue, but they do not affect whether the decltype results of their unparenthesized member variables are reference types or not.

If *expression* declared in decltype(*expression*) is a parenthesized nonstatic non-reference class member variable, the constant or volatile type qualifier of the parent object expression or pointer expression of *expression* contributes to the result of decltype(*expression*). Similarly, the lvalueness or rvalueness of the object expression or the pointer expression affects the result of decltype(*expression*).

Example:

```
struct Foo{
    int x;
};
```

```
int main(){
    int i = 1;
    struct Foo f;
    const struct Foo g = {0};
    volatile struct Foo* h = &f;
    struct Foo func();

    decltype((g.x)) var1 = i;         // const int&
    decltype((h->x)) var2 = i;        // volatile int&
    decltype((func().x)) var3 = 1;    // int
    return 0;
}
```

In this example, the result of `decltype((g.x))` inherits the constant qualifier of the object expression `g`. Similarly, the result of `decltype((h->x))` inherits the volatile qualifier of the pointer expression `h`. The object expression `g` and the pointer expression `h` are lvalues, so `decltype((g.x))` and `decltype((h->x))` are reference types. The object expression `func()` is an rvalue, so `decltype((func().x))` is a nonreference type.

If you use the built-in operators .* or ->* within a decltype(*expression*), the constant or volatile type qualifier of the parent object expression or pointer expression of *expression* contributes to the result of decltype(*expression*), regardless of whether *expression* is a parenthesized or an unparenthesized structure member variable. Similarly, the lvalueness or rvalueness of the object expression or the pointer expression affects the result of decltype(*expression*).

Example:

```
class Foo{
    int x;
};
int main(){
   int i = 0;
   Foo f;
   const Foo & g = f;
   volatile Foo* h = &f;
   const Foo func();

   decltype(f.*&Foo::x)  var1 = i;        // int&, f is an lvalue
   decltype(g.*&Foo::x) var2 = i;         // const int&, g is an lvalue
   decltype(h->*&Foo::x) var3 = i;        // volatile int&, h is an lvalue
   decltype((h->*&Foo::x)) var4 = i;      // volatile int&, h is an lvalue
   decltype(func().*&Foo::x) var5 = 1;    // const int, func() is an rvalue
   decltype((func().*&Foo::x)) var6 = 1;  // const int, func() is an rvalue
   return 0;
}
```

Side effects and decltype

If you use decltype(*expression*) to get a type, additional operations in the decltype parenthetical context can be performed, but they do not have side effects outside of the decltype context. Consider the following example:

```
int i = 5;
static const decltype(i++) j = 4;    // i is still 5
```

The variable *i* is not increased by 1 outside of the decltype context.

There are exceptions to this rule. In the following example, because the expression given to decltype must be valid, the compiler has to perform a template instantiation:

```
template <int N>
struct Foo{
    static const int n=N;
};
int i;

decltype(Foo<101>::n,i) var = i;     // int&
```

In this example, `Foo` template instantiation occurs, even though `var` is only determined by the type of the variable `i`.

Redundant qualifiers and specifiers with decltype

Because decltype(*expression*) is considered syntactically to be a type specifier, the following redundant qualifiers or specifiers are ignored:

- constant qualifiers
- volatile qualifiers
- & specifiers

The following example demonstrates this case:

```
int main(){
    int i = 5;
    int& j = i;
    const int k = 1;
    volatile int m = 1;

    // int&, the redundant & specifier is ignored
    decltype(j)&  var1 = i;

    // const int, the redundant const qualifier is ignored
    const decltype(k) var2 = 1;

    // volatile int, the redundant volatile qualifer is ignored
    volatile decltype(m) var3;
    return 0;
}
```

Note: The functionality of ignoring the redundant & specifiers in decltype(*expression*) is not supported in the current C++11 standard, but it is implemented in this compiler release.

Template dependent names and decltype

Without using the decltype feature, when you pass parameters from one function to another function, you might not know the exact types of the results that are passed back. The decltype feature provides a mechanism to generalize the return types easily. The following program shows a generic function that performs the multiplication operation on some operands:

```
struct Math{
    template <typename T>
    static T mult(const T& arg1, const T& arg2){
        return arg1 * arg2;
    }
};
```

If `arg1` and `arg2` are not the same type, the compiler cannot deduce the return type from the arguments. You can use the decltype feature to solve this problem, as shown in the following example:

```
struct Foo{
   template<typename T, typename U>
   static decltype((*(T*)0)*(*(U*)0)) mult(const T& arg1, const U& arg2)
   {
       return arg1 * arg2;
   }
};
```

In this example, the return type of the function is the type of the multiplication result of the two template-dependent function parameters.

The typeof operator and decltype

IBM The decltype feature is similar to the existing *typeof* feature. One difference between these two features is that decltype accepts only an expression as its operand, while typeof can also accept a type name. Consider the following example:

```
__typeof__(int)  var1;     // okay
decltype(int) var2;        // error
```

In this example, `int` is a type name, so it is invalid as the operand of decltype.

Note: `__typeof__` is an alternate spelling of typeof.

IBM

Related reference:

"Keywords" on page 13

"Name binding and dependent names" on page 480

"C++11 compatibility" on page 640

"Lvalues and rvalues" on page 141

"References (C++ only)" on page 107

The constexpr specifier (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

The C++11 standard introduces a new keyword `constexpr` as a declaration specifier. You can apply the `constexpr` specifier only to the following contexts:

- The definition of a variable
- The declaration of a function or function template
- The declaration of a static data member

For example:

```
constexpr int i = 1;     // OK, definition
constexpr int f1();      // OK, function declaration, but must be defined before use
```

If you declare a function that is not a constructor with a `constexpr` specifier, that function is a `constexpr` function. Similarly, if you declare a constructor with a

constexpr specifier, that constructor is a constexpr constructor. Both constexpr functions and constexpr constructors are implicitly inline. For example:

```
struct S {
  constexpr S(int i) : mem(i) { }  // OK, declaration of a constexpr constructor
  private:
    int mem;
};
constexpr S s(55);  // OK, invocation of a constexpr constructor
```

If any declaration of a function or function template is specified with constexpr, all its declarations must contain the constexpr specifier. For example:

```
constexpr int f1();  // OK, function declaration
int f1() {           // Error, the constexpr specifier is missing
  return 55;
}
```

Function parameters cannot be declared with the constexpr specifier. The following example demonstrates this:

```
constexpt int f4(constexpr int);  //Error
```

A constexpr specifier used in an object declaration declares the object as const. Such an object must be of a literal type and initialized. If it is initialized by a constructor call, that call must be a constant expression. Otherwise, if a constexpr specifier is used in a reference declaration, every full expression that appears in its initializer must be a constant expression. Each implicit conversion used in converting the initializer expressions and each constructor call used for the initialization must be valid in a constant expression. For example:

```
constexpr int var;       // Error, var is not initialized
constexpr int var1 = 1;  // OK

void func() {
  var1 = 5;  //Error, var1 is const
}

struct L {
  constexpr L() : mem(55) { }
  constexpr L(double d) : mem((int)d) { }
  L(int i) : mem(i) { }
  operator int() { return mem; }
private:
  int mem;
};

// Error, initializer involves a non-constexpr constructor.
constexpr L var2(55);

double var3 = 55;

// Error, initializer involves a constexpr constructor with non-constant argument
constexpr L var4(var3);

// Error, involves conversion that uses a non-constexpr conversion function
constexpr int var5 = L();
```

A constexpr specifier for a nonstatic member function that is not a constructor declares that member function to be const. The class of that constexpr member function must be a literal type. In the following example, the class NL is a non-literal type because it has a user-provided destructor.

```
struct NL {
  constexpr int f(){       //error, enclosing class is not a literal type
     return 55;
  }
  ~NL() { }
};
```

A call to a `constexpr` function produces the same result as a call to an equivalent non-`constexpr` function, except that a call to a `constexpr` function can appear in a constant expression.

The `main` function cannot be declared with the `constexpr` specifier.

Related reference:

"Literals" on page 19

"Constexpr functions (C++11)" on page 313

"Constexpr constructors (C++11)" on page 411

"Generalized constant expressions (C++11)" on page 149

Compatibility of arithmetic types (C only)

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

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.

Type qualifiers

A type qualifier is used to refine the declaration of a variable, a function, and parameters, by specifying whether:

- The value of an object can be changed
- The value of an object must always be read from memory rather than from a register
- More than one pointer can access a modifiable memory address

z/OS XL C/C++ recognizes the following type qualifiers:

- `const`
- `restrict`
- `volatile`

z/OS XL C/C++ includes the following additional type qualifiers to meet the special needs of the z/OS environment:

- `__callback`
- C `__far` C
- `__ptr32`
- C `__ptr64` C

Standard C++ refers to the type qualifiers `const` and `volatile` as *cv-qualifiers*. In both languages, the cv-qualifiers are only meaningful in expressions that are lvalues.

When the `const` and `volatile` keywords are used with pointers, the placement of the qualifier is critical in determining whether it is the pointer itself that is to be qualified, or the object to which the pointer points. For a pointer that you want to qualify as `volatile` or `const`, 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 and qualifier can be in any order, provided that the qualifier does not follow the `*` operator. For example, for a pointer to a `volatile` data object:

```
volatile int *x;         /* x is a pointer to a volatile int  */
```

or

```
int volatile *x;         /* x is a pointer to a volatile int  */
```

For a pointer to a `const` data object:

```
const int *y;            /* y is a pointer to a const int  */
```

or

```
int const *y;            /* y is a pointer to a const int  */
```

The following examples contrast the semantics of these declarations:

Declaration	Description
`const int * ptr1;`	Defines a pointer to a constant integer: the value pointed to cannot be changed.
`int * const ptr2;`	Defines a constant pointer to an integer: the integer can be changed, but `ptr2` cannot point to anything else.
`const int * const ptr3;`	Defines a constant pointer to a constant integer: neither the value pointed to nor the pointer itself can be changed.

You can put more than one qualifier on a declaration, and the compiler ignores duplicate type qualifiers. This is a C99 language feature. C++ To be compatible with C99, the z/OS XL C/C++ compiler supports it as an IBM extension. C++

A type qualifier cannot apply to user-defined types, but only to objects created from a user-defined type. Therefore, the following declaration is illegal:

```
volatile struct omega {
                        int limit;
                        char code;
                      }
```

However, if a variable or variables are declared within the same definition of the type, a type qualifier can be applied to the variable or variables by placing it at the beginning of the statement or before the variable declarator or declarators. Therefore:

```
volatile struct omega {
                      int limit;
                      char code;
                      } group;
```

provides the same storage as:

```
struct omega {
              int limit;
              char code;
              }  volatile group;
```

In both examples, the `volatile` qualifier only applies to the structure variable `group`.

When type qualifiers are applied to a structure, C++ class C++ , or union variable, they also apply to the members of the structure, class or union.

Related reference:

"Pointers" on page 100

"Constant and volatile member functions" on page 360

The __callback type qualifier

The keyword `__callback` is a qualifier that can be applied only to a function pointer type. The qualifier instructs the compiler to generate extra code in the call sites to assist the call, and thus allows the function pointer to point to either XPLINK or non-XPLINK functions. Under normal circumstances, a non-XPLINK function pointer is incompatible with XPLINK compilation units.

The keyword can appear in the declarator part of a function pointer declaration, wherever a cv-qualifier can appear. For example,

```
int (*__callback foo)(int);
```

declares `foo` to be a function pointer that might point to non-XPLINK functions. `foo` will then have fewer restrictions on what it can reference and can thus be used with XPLINK compilation units.

XPLINK and non-XPLINK compilation units cannot be statically bound; the two linkages can be mixed only across DLL boundaries. Moreover, a function pointer that points to a non-XPLINK function cannot be used in XPLINK DLLs unless the pointer is passed across the boundary explicitly as a function argument. The `__callback` qualifier relaxes the latter restriction, at the expense of extra code sequences in the call site.

Semantically, the `__callback` keyword is a language extension that has a single effect: to instruct the compiler to generate assistance code. It does not take part in type definition. The keyword also has no effect on the following:

- Type (such as in overload resolution).
- Name mangling.
- Allocation of the pointer object in memory.

It is the responsibility of the programmer to make sure that the function pointer is appropriately `__callback`-qualified for all call sites that require it.

The atomic qualifier (C11)

The `_Atomic` qualifier designates an atomic type. The type modified by the `_Atomic` qualifier cannot be an array type or a function type.

The properties associated with qualified types are meaningful only for expressions that are lvalues.

If you modify a type with the _Atomic qualifier and other qualifiers, the resulting type is the so-qualified atomic type. For example, the following statement declares a pointer to a volatile-qualified atomic type:

```
_Atomic volatile int *p;
```

The const type qualifier

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 left side of an assignment statement.

C A const object cannot be used in constant expressions. A global const object without an explicit storage class is considered extern by default. C

C++ In C++, all const declarations must have initializers, except those referencing externally defined constants. 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 global const object without an explicit storage class is considered static by default, with internal linkage.

```
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++ */
```

Because its linkage is assumed to be internal, a const object can be more easily defined in header files in C++ than in C. C++

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.

Related reference:

"The #define directive" on page 507

"The this pointer" on page 363

The __far type qualifier (C only)

When the METAL option is in effect, you can use the __far keyword to qualify a pointer type so that it can access additional data spaces in access-register (AR) mode. The upper half of the pointer contains the access-list-entry token (ALET), which identifies the secondary virtual address space you want to access. The lower half the pointer is the offset within the secondary virtual address space. The size of a __far-qualified pointer is increased to 8 bytes in 31-bit mode and 16 bytes in 64-bit mode. In 31-bit mode, the upper 4 bytes contain the ALET, and the lower 4 bytes is the address within the data space. In 64-bit mode, bytes 0-3 are unused, bytes 4-7 are the ALET, and bytes 8-15 are the address within the data space.

The __far keyword must appear in the declarator part of a pointer declaration, wherever a cv-qualifier can be used. For example,

```
int * __far p;
```

declares p to be a __far pointer to int.

__far pointers can appear in global scope and function scope, in simple assignment and in implicit assignment via function parameter passing. However, if they are used inside a function in operations that access the data space, such as dereferencing, the function must be in AR mode (that is, with the ARMODE compiler option in effect, or qualified with the armode function attribute).

A normal pointer can be converted to a __far pointer explicitly through typecasting or implicitly through assignment. The ALET of the __far pointer is set to zero. A __far pointer can be explicitly converted to a normal pointer through typecasting; the normal pointer keeps the offset of the __far pointer and the ALET is lost. A __far pointer cannot be implicitly converted to a normal pointer.

Pointer arithmetic is supported for __far pointers, with the ALET part being ignored. If the two ALETs are different, the results may have no meaning.

Two __far pointers can be compared for equality and inequality using the == and != operators. The whole pointer is compared. To compare for equality of the offset only, use the built-in function to extract the offset and then compare. To compare for equality of the ALET only, use the built-in function to extract the ALET and then compare. For more information on the set of built-in functions that operate on __far pointers, see *z/OS XL C/C++ Programming Guide*.

Two __far pointers can be compared using the >, < , >=, and <= relational operators. The ALET parts of the pointers are ignored in this operation. There is no ordering between two __far pointers if their ALETs are different, and between a NULL pointer and any __far pointers. The result is meaningless if they are compared using relational operators.

When a __far pointer and a normal pointer are involved in an operation, the normal pointer is implicitly converted to __far before the operation. There is unspecified behavior if the ALETs are different. For example:

```
int * __far p;
int * __far q;
ptrdiff_t  chunk;
...

if (p == q) {
    p = p + 1024;
}

if (p < q) {
    chunk = q - p;
}
else {
    chunk = p - q;
}
```

The result of the & (address) operator is a normal pointer, except for the following cases:

- If the operand of & is the result of an indirection operator (*), the type of & is the same as the operand of the indirection operator.
- If the operand of & is the result of the arrow operator (->, structure member access), the type of & is the same as the left operand of the arrow operator.

For example:

```
int * __far p;
int * __far q;
...

q  =  &(*(p+2));  // result of & is a __far pointer; the ALET is the same as p.

struct  S {
   int b;
} * __far r;
...
q = & r->b;  // result of & is a __far pointer; the ALET is the same as r.
```

For more information on ARMODE and METAL compiler options, see ARMODE and METAL compiler options in the *z/OS XL C/C++ User's Guide*.

Related reference:

"armode | noarmode (C only)" on page 244

The __ptr32 type qualifier

The keyword __ptr32 is a qualifier that can be applied to a pointer type to constrain its size to 32 bits. This language extension is provided to facilitate porting structures with pointer members from 31- to 64-bit mode. The qualifier is accepted and ignored in 31-bit mode.

The size of a pointer type doubles to 64 bits in 64-bit mode. Doubling the size of a pointer changes the layout of a structure that contains pointer members. If the object referenced by a pointer member resides within a 31-bit addressing space, constraining the pointer to 32 bits can reduce some of the unexpected effects of moving to 64-bit mode.

The __ptr32 keyword can appear in the declarator part of a pointer declaration, wherever a cv-qualifier can be used. For example,

```
int * __ptr32 p;
```

declares p to be a 32-bit pointer to int.

```
int * __ptr32 *q;
```

declares q to be a 64-bit pointer to a 32-bit pointer to int.

```
int * __ptr32 const r;
```

declares r to be a const 32-bit pointer.

Pointers with external linkage must be __ptr32-qualified consistently across all compilation units. If a pointer is declared 31-bit in one compilation unit and 64-bit in another, the behavior is undefined.

Assignment of 32-bit and 64-bit pointers to each other is permitted. The compiler generates an implicit conversion or truncates without emitting a diagnostic.

Note: The terms *31-bit mode* and *32-bit mode* are used interchangeably when there is no ambiguity. The term *32-bit mode* is commonly used in the industry to refer to a class of machines, to which z/OS in 31-bit mode belongs. Strictly speaking, *31-bit mode* refers to the addressing mode of the architecture, and *32 bits* refers to the size of the pointer type. In z/OS 31-bit addressing mode, the size of a pointer is four bytes. However, the high-order bit is reserved for system use, and is not used to form the address. The addressing range in this mode is therefore 2 gigabytes. In

64-bit mode, the size of a pointer is eight bytes, and all 64 bits participate in addressing. When a __ptr32 pointer is dereferenced, a 64-bit address is formed by filling the 33 missing high-order bits with zeros. The program using that address should make sure it is valid within the address space of the application.

The __ptr64 type qualifier (C only)

The keyword __ptr64 is a qualifier that can be applied to a pointer type to constrain its size to 64 bits. When you need to switch addressing mode (AMODE) between programs, this language extension enables the handling of a 64-bit pointer by an AMODE 31 function without dereferencing it, for example, passing it as a parameter or receiving it as a return value.

Note: The __ptr64 qualifier can be used only when the METAL compiler option is specified.

The __ptr64 keyword can be used only to qualify a pointer type. For example,

```
int *__ptr64 p; /* 64-bit pointer */
int *r; /* 32-bit pointer, default to the model's size */
int *__ptr64 const q; /* 64-bit const pointer */
int *__far __ptr64 s; /* 64-bit far pointer */
```

For further information on the METAL compiler option, see *z/OS XL C/C++ User's Guide.* For further information on AMODE switching, see *z/OS Metal C Programming Guide and Reference.*

The restrict type qualifier

This type qualifier is introduced in the C99 standard. C++ The z/OS XL C/C++ compiler supports it as an IBM extension. C++

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 (or __restrict or __restrict__)[1] type qualifier can be applied to a pointer type to form a restrict-qualified pointer. During the execution of the block that is associated with the declaration of an object that provides a way to designate a restrict-qualified pointer, the memory addressed via the restrict-qualified pointer cannot be modified or can be accessed only via this pointer if the pointer does not point to a const-qualified type. 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.

Notes:

1. The `restrict` qualifier is represented by the following keywords (all have the same semantics):
   - The `restrict` keyword is recognized in C, under compilation with **c99** or the **LANGLVL(STDC99)** or **LANGLVL(EXTC99)** options, and in C++ under the **LANGLVL(EXTENDED)** or **KEYWORD(RESTRICT)** options. IBM The `__restrict` and `__restrict__` keywords are recognized in both C, at all language levels, and C++, at **LANGLVL(EXTENDED)**. IBM

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 or by another program). This prevents the compiler from optimizing code referring to the object by storing the object's value in a register and re-reading it from there, rather than from memory, where it may have changed.

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. C

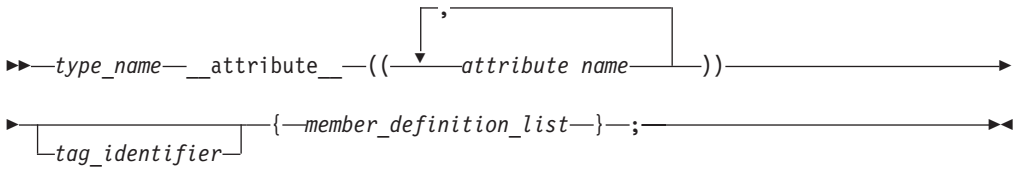
An item can be both `const` and `volatile`. In this case the item cannot be legitimately modified by its own program but can be modified by some asynchronous process.

Type attributes (IBM extension)

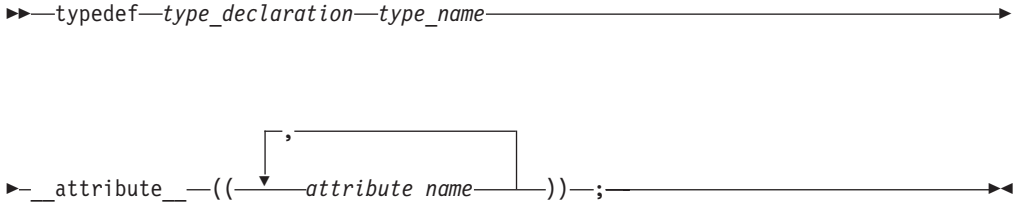
Type attributes are language extensions that allow you to use named attributes to specify special properties of user-defined types. Type attributes apply to the definitions of user-defined types, such as structures, unions, enumerations, C++ and classes C++ . Any objects that are declared as having that type will have the attribute applied to them.

A type attribute is specified with the keyword `__attribute__` followed by the attribute name and any additional arguments the attribute name requires. Although there are variations, the syntax of a type attribute is of the general form:

Type attribute syntax



Type attribute syntax — typedef declarations



For unsupported attribute names, the z/OS XL C/C++ compiler issues diagnostics and ignores the attribute specification. Multiple attribute names can be specified in the same attribute specification.

The following type attributes are supported:

- "The amode31 | amode64 type attribute (C only)"
- "The armode | noarmode type attribute (C only)" on page 94
- "The may_alias type attribute" on page 94

Related reference:

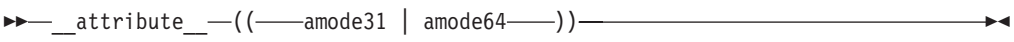
"Variable attributes (IBM extension)" on page 125

"Function attributes (IBM extension)" on page 242

The amode31 | amode64 type attribute (C only)

For use with the METAL compiler option, the `amode31` type attribute allows you to define a `typedef` of a function or function pointer type to operate in addressing mode (AMODE) 31 and the `amode64` type attribute allows you to define a `typedef` of a function or function pointer type to operate in AMODE 64.

amode31 | amode64 function attribute syntax



The following example declares a `typedef` of function pointer `foo` that is in AMODE 64:

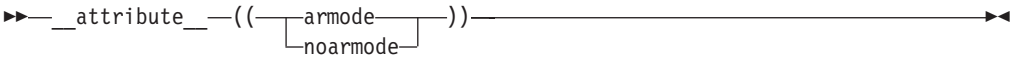
```
typedef void (*foo)(int) __attribute__ ((amode64));
```

For information on the METAL compiler option, see the METAL compiler option description in *z/OS XL C/C++ User's Guide.* For information on AMODE switching, see *z/OS Metal C Programming Guide and Reference.*

The armode | noarmode type attribute (C only)

For use with the METAL compiler option, the `armode` type attribute allows you to define a typedef of function or function pointer type as operating in access-register (AR) mode. AR mode allows a C function to access multiple additional data spaces, and manipulate more data in memory.

armode function attribute syntax



Functions in AR mode can call functions not in AR mode, and vice versa.

The following example declares a `typedef` of function pointer `foo` that is in AR mode, and then declares `bar` as a function that passes function pointer `foo` as a parameter:

```
typedef void (*foo) (int) __attribute__((armode));
void bar (foo);
```

The attribute overrides the default setting of the ARMODE compiler option for the specified type. Note that this attribute is only supported when the METAL compiler option is in effect.

For more information on ARMODE and METAL compiler options, see ARMODE and METAL compiler options in the *z/OS XL C/C++ User's Guide*.

Related reference:

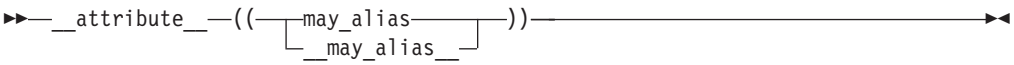
"armode | noarmode (C only)" on page 244

"The __far type qualifier (C only)" on page 88

The may_alias type attribute

You can specify the `may_alias` type attribute for a type so that lvalues with dereferencing operator of the type can alias objects of any type, similar to a `char` type. Types with the `may_alias` attribute are not subject to type-based aliasing rules.

may_alias type attribute syntax



You can specify the may_alias type attribute in the following ways:

```
struct __attribute__((__may_alias__)) my_struct {} *ps;
typedef long __attribute__((__may_alias__)) t_long;
typedef struct __attribute__((__may_alias__)) my_struct {} t_my_struct;
```

Instead of specifying the **NOANSIALIAS** option, you can alternatively specify the may_alias type attribute for a type to violate the ANSI aliasing rules when compiling expressions that contain lvalues of that type. For example:

```
#define __attribute__(x)     // Invalidates all __attribute__ declarations
typedef long __attribute__((__may_alias__)) t_long;

int main (void){
  int i = 42;
  t_long *pa = (t_long *) &i;
  *pa = 0;
  if (i == 42)
    return 1;
  return 0;
}
```

If you compile this code with the **ANSIALIAS** option at a high optimization level, such as **-O3**, the executable program returns 1. Because the lvalue *pa is of type long, according to the ANSI aliasing rules, the assignment to lvalue *pa cannot modify the value of i, which is of type int.

If you remove the #define __attribute__(x) statement and compile the code with the same options as before, the executable program returns 0. Because the type of *pa is long __attribute__((__may_alias__)), *pa can alias any other object of any type, and the assignment to lvalue *pa can modify the value of i to 0.

The usage of the may_alias type attribute can result in less conservative aliasing relationships and provide more optimization opportunities compared to usage of compiler option **ANSIALIAS**.

C This attribute is supported at the EXTC89, EXTC99, EXTENDED, and EXTC1X language levels. C

C++ This attribute is supported at the EXTENDED and EXTENDED0X language levels. C++

Chapter 4. Declarators

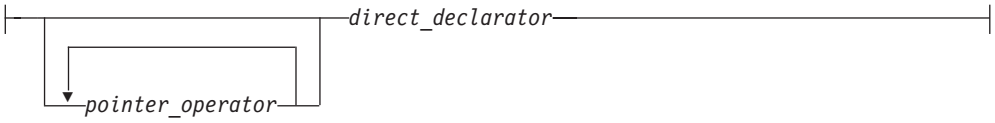
This section continues the discussion of data declarations and includes information on type names, pointers, arrays, C++ references C++ , initializers, and variable attributes.

Overview of declarators

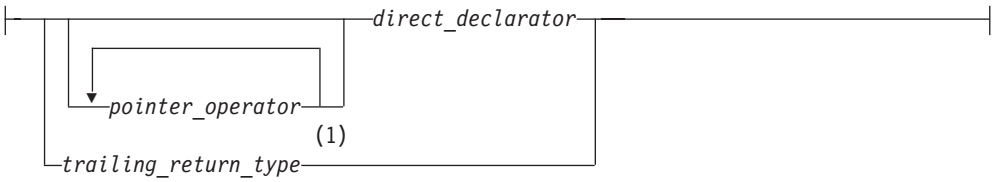
A declarator declares an object, function, or C++ reference C++ as part of a declaration.

A declarator has the following form:

Declarator syntax (C only):



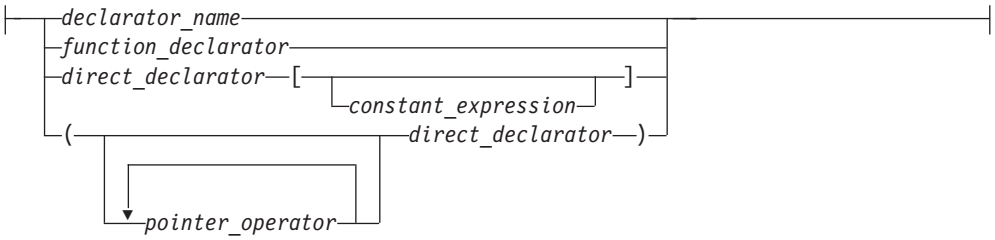
Declarator syntax (C++ only):



Notes:

1 C++11

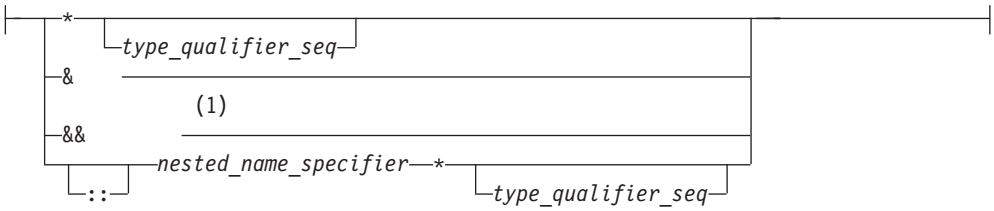
Direct declarator:



Pointer operator (C only):

Declarator name (C only):

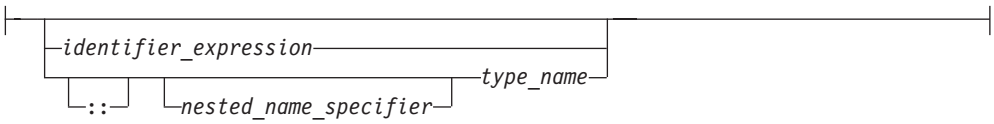
Pointer operator (C++ only):



Notes:

1 C++11

Declarator name (C++ only):



Notes:

- The *type_qualifier_seq* represents one or a combination of type qualifiers. For the details of type qualifiers, see "Type qualifiers" on page 85.
- C++ A *nested_name_specifier* is a qualified identifier expression. An *identifier_expression* can be a qualified or unqualified identifier. C++

For the details of function declarators, see "Function declarators" on page 235. C++11 For the details of trailing return types, see "Trailing return type (C++11)" on page 239. C++11

The following types are known as *derived declarator* types, and are therefore discussed in this section:

- "Pointers" on page 100
- "Arrays" on page 104
- "References (C++ only)" on page 107

z/OS XL C/C++ includes two additional qualifiers, which are described in "Declarator qualifiers" on page 124.

IBM In addition, for compatibility with GNU C and C++, z/OS XL C/C++ allows you to use *variable attributes* to modify the properties of data objects. As they are normally specified as part of the declarator in a declaration, they are described in "Variable attributes (IBM extension)" on page 125. IBM

Related reference:

"Initializers" on page 108

"Type qualifiers" on page 85

Examples of declarators

The following table indicates the declarators within the declarations:

Declaration	Declarator	Description
`int owner;`	`owner`	`owner` is an integer data object.
`int *node;`	`*node`	`node` is a pointer to an integer data object.
`int names[126];`	`names[126]`	`names` is an array of 126 integer elements.
`volatile int min;`	`min`	`min` is a volatile integer.
`int * volatile volume;`	`* volatile volume`	`volume` is a volatile pointer to an integer.
`volatile int * next;`	`*next`	`next` is a pointer to a volatile integer.
`volatile int * sequence[5];`	`*sequence[5]`	`sequence` is an array of five pointers to volatile integer data objects.
`extern const volatile int clock;`	`clock`	`clock` is a constant and volatile integer with static storage duration and external linkage.
`int * __far p;`	`* __far p`	p is a `__far` pointer to an integer

Related reference:

"Type qualifiers" on page 85

"Array subscripting operator [ ]" on page 171

"Scope resolution operator :: (C++ only)" on page 148

"Function declarators" on page 235

Type names

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 type name 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.

Table 18. Type names

Syntax	Description
`int *[5]`	array of 5 pointers to `int`
`int (*)[5]`	pointer to an array of 5 integers
`int (*)[*]`	pointer to an variable length array of an unspecified number of integers
`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. C++

Related reference:

"Operator precedence and associativity" on page 190

"Examples of expressions and precedence" on page 193

"The typename keyword" on page 482

"Parenthesized expressions ( )" on page 145

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.

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 z/OS XL C compiler supports only the pointers that are obtained in one of the following ways:

- Directly from the return value of a library function which returns a pointer
- As an address of a variable
- From constants that refer to valid addresses or from the NULL constant
- Received as a parameter from another C function
- Directly from a call to a service in the z/OS IBM Language Environment® that allocates storage, such as CEEGTST

Any bitwise manipulation of a pointer can result in undefined behavior. C

Note that the placement of the type qualifiers `volatile` and `const` affects the semantics of a pointer declaration. If either of the qualifiers appears before the `*`, the declarator describes a pointer to a type-qualified object. If either of the qualifiers appears between the `*` and the identifier, the declarator describes a type-qualifed pointer.

The following table provides examples of pointer declarations.

Table 19. Pointer declarations

Declaration	Description
`long *pcoat;`	`pcoat` is a pointer to an object having type `long`
`extern short * const pvolt;`	`pvolt` is a constant pointer to an object having type `short`
`extern int volatile *pnut;`	`pnut` is a pointer to an `int` object having the `volatile` qualifier
`float * volatile psoup;`	`psoup` is a `volatile` pointer to an object having type `float`
`enum bird *pfowl;`	`pfowl` is a pointer to an enumeration object of type `bird`
`char (*pvish)(void);`	`pvish` is a pointer to a function that takes no parameters and returns a `char`
C++11 `nullptr_t pnull;`	`pnull` is a null pointer that does not point to any valid object or function. C++11

Related reference:

"Type qualifiers" on page 85
"Initialization of pointers" on page 116
"Compatibility of pointers (C only)" on page 103
"Pointer conversions" on page 137
"Address operator &" on page 154
"Indirection operator *" on page 155
"Pointers to functions" on page 257

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.

Related reference:

"Increment operator ++" on page 151

"Arrays" on page 104

"Decrement operator --" on page 152

Chapter 6, "Expressions and operators," on page 141

Type-based aliasing

The compiler follows the type-based aliasing rule in the C and C++ standards when the ANSIALIAS option is in effect (which it is by default). This rule, also known as the ANSI aliasing rule, states that a pointer can only be dereferenced to an object of the same type or a compatible type. [1]

---

1. The C Standard states that an object shall have its stored value accessed only by an lvalue that has one of the following types:
   - the declared type of the object,
   - a qualified version of the declared type of the object,
   - a type that is the signed or unsigned type corresponding to the declared type of the object,
   - a type that is the signed or unsigned type corresponding to a qualified version of the declared type of the object,
   - an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or
   - a character type

   The C++ standard states that if a program attempts to access the stored value of an object through an lvalue of other than one of the following types, the behavior is undefined:
   - the dynamic type of the object,

The common coding practice of casting a pointer to an incompatible type and then dereferencing it violates this rule. (Note that `char` pointers are an exception to this rule) Refer to the description of the ANSIALIAS option in the *z/OS XL C/C++ User's Guide* for additional information.

The compiler uses the type-based aliasing information to perform optimizations to the generated code. Contravening the type-based aliasing rule can lead to unexpected behavior, as demonstrated in the following example:

```
int *p;
double d = 0.0;

int *faa(double *g);         /* cast operator inside the function   */

void foo(double f) {
    p = faa(&f);             /* turning &f into an int ptr     */
    f += 1.0;                /* The optimizer might move the  */
                             /* assignment after the printf statement. */
    printf("f=%x\n", *p);
}

int *faa(double *g) { return (int*) g; }   /* questionable cast;       */
                                           /* the function can be in   */
                                           /* another translation unit */

int main() {
       foo(d);
 }
```

In the above `printf` statement, `*p` cannot be dereferenced to a `double` under the ANSI aliasing rule. The compiler determines that the result of `f += 1.0` does not affect the value of `*p`. Thus, the optimizer might move the assignment after the `printf` statement. If you compile the above example with optimization enabled, the `printf` statement might output `0` (zero).

Related reference:

"The may_alias type attribute" on page 94

"The reinterpret_cast operator (C++ only)" on page 179

Compatibility of pointers (C only)

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.

The following example shows compatible declarations for the assignment operation:

- a cv-qualified version of the dynamic type of the object,
- a type that is the signed or unsigned type corresponding to the dynamic type of the object,
- a type that is the signed or unsigned type corresponding to a cv-qualified version of the dynamic type of the object,
- an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union),
- a type that is a (possible cv-qualified) base class type of the dynamic type of the object,
- a char or unsigned char type.

```
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 */
```

z/OS Packed and nonpacked objects have different memory layouts. Consequently, a pointer to a packed structure or union is incompatible with a pointer to a corresponding nonpacked structure or union. As a result, comparisons and assignments between pointers to packed and nonpacked objects are not valid.

You can, however, perform these assignments and comparisons with type casts. In the following example, the cast operation lets you compare the two pointers, but you must be aware that `ps1` still points to a nonpacked object:

```
int main(void)
{
   _Packed struct ss *ps1;
   struct ss         *ps2;
      ...
   ps1 = (_Packed struct ss *)ps2;
      ...}
```

z/OS

Related reference:

"The _Packed qualifier (C only)" on page 124

Arrays

An *array* is a collection of objects of the same data type, allocated contiguously in memory. 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 not supported by either language.

The type of an array 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.

C++ Array elements may not be of reference type or of an abstract class type. C++

The array declarator contains an identifier followed by an optional *subscript declarator*. An identifier preceded by an asterisk (*) is an array of pointers.

Array subscript declarator syntax

The *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 *variable-length array*, as described in "Variable length arrays" on page 106. C

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]
```

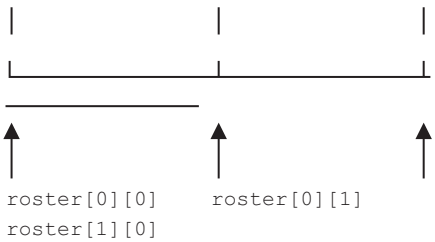
The following example defines a two-dimensional array that contains six elements of type `int`:

```
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.

Related reference:

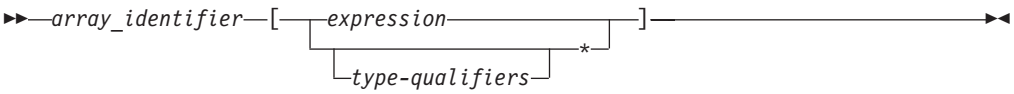
"Array subscripting operator [ ]" on page 171

"Initialization of arrays" on page 116

Variable length arrays

A variable length array, which is a C99 feature, is an array of automatic storage duration whose length is determined at run time. C++ The z/OS XL C/C++ compiler supports this feature as an IBM extension. C++

Variable length array declarator syntax



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.

Note: C++ In C++ applications, storage allocated for use by variable length arrays is not released until the function they reside in completes execution. C++

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` statement. 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.

IBM C++ The C++ extension does not include support for references to a variable length array type; neither might a function parameter be a reference to a variable length array type. C++ IBM

Related reference:

Flexible array members

Compatibility of arrays (C only)

Two compatible array types must have compatible element types. In addition, if each array has a size specifier that is an integer constant expression, both size specifiers must have the same constant value. For example, the types of the following two arrays are not compatible:

```
char ex1[25];
const char ex2[25];
```

The composite type of two compatible array types is an array with the composite element type. The composite type of two compatible arrays is determined by the following rules:

1. If one of the original types is an array of known constant size, the composite type is an array of that size. For example:

   ```
   // The composite type is char [42].
   char ex3[];
   char ex4[42];
   ```

2. Otherwise, if one of the original types is a variable length array, the composite type is that type.
3. C11 Otherwise, if one of the original types is a variable length array whose size is specified by an expression that is not evaluated, the behavior is undefined.
4. Otherwise, if one of the original types is a variable length array whose size is specified by an expression that is already evaluated, the composite type is a variable length array of that size. For example:

   ```
   // The composite type is int [n].
   int ex5[];
   int ex6[n]; // The value of n is determined
   ```

5. Otherwise, if one of the original types is a variable length array of unspecified size, the composite type is a variable length array of unspecified size.
6. Otherwise, if both the original types are arrays of unknown size, the composite type is an array of unknown size. For example:

   ```
   // The composite type is int [].
   int ex7[];
   int ex8[];
   ```

C11

Related reference:

"External linkage" on page 8

References (C++ only)

A *reference* is an alias or an alternative name for an object or function. All operations applied to an object reference act on the object to which the reference refers. The address of a reference is the address of the aliased object or function.

An lvalue reference type is defined by placing the reference modifier & or bitand after the type specifier. C++11 An rvalue reference type is defined by placing the reference modifier && or and after the type specifier. For the details of rvalue references, see **Using rvalue references (C++11)**. C++11 Reference types include both lvalue reference C++11 and rvalue reference C++11 types.

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 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.

The following types of references are invalid:

- References to NULL
- References to void
- References to invalid objects or functions
- References to bit fields
- References to references C++11 except with reference collapsing. See "Reference collapsing (C++11)" on page 194 for more information. C++11

You also cannot declare arrays of references, pointers to references, and cv-qualifiers on references. If cv-qualifiers are introduced through a typedef or template argument deduction, the cv-qualifiers are ignored.

For information on references to functions, see "Pointers to functions" on page 257.

Related reference:

"Initialization of references (C++ only)" on page 119

"Pointers" on page 100

"Address operator &" on page 154

"Pass by reference (C++ only)" on page 252

Initializers

An *initializer* specifies an initial value to a data object and is optional in a data declaration. Whether an initializer is valid for a particular declaration depends on the type and storage class of the object to be initialized.

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;
```

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

C++ You can initialize variables at namespace scope with nonconstant expressions. C++ C You cannot initialize variables at global scope with nonconstant expressions. C

Related reference:

"Using class objects" on page 348

Initialization and storage classes

This topic includes descriptions of the following:

- Initialization of automatic variables
- Initialization of static variables
- Initialization of external variables
- Initialization of register variables

Initialization of automatic variables

You can initialize any `auto` variable except function 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.

Note: C++11 In C++11, the keyword `auto` is no longer used as a storage class specifier. Instead, it is used as a type specifier. The compiler deduces the type of an `auto` variable from the type of its initializer expression. For more information, see "The auto type specifier (C++11)" on page 76. C++11

Initialization of static variables

You can 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, unless it is a pointer, in which case it will be initialized to `NULL`.

C 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. C

C++ A static variable in a block can be dynamically initialized when the flow of control passes through its definition in a block for the first time. Dynamic initialization of a static variable can occur with non-constant expressions. A static object of class type will use the default constructor if you do not initialize it. C++

Initialization of external variables

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:

- C Appear as part of the definition, and the initial value must be described by a constant expression; C
- C++ Appear as part of the definition. C++
- 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.

Initialization of register variables

You can initialize any register object except function 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.

Related reference:

"The auto storage class specifier" on page 49

"The static storage class specifier" on page 49

"The extern storage class specifier" on page 51

"The register storage class specifier" on page 52

Designated initializers for aggregate types (C only)

Designated initializers, a C99 feature, are supported for aggregate types, including arrays, structures, and unions. A designated initializer, or *designator*, points out a particular element to be initialized. A *designator list* is a comma-separated list of one or more designators. A designator list followed by an equal sign constitutes a *designation*.

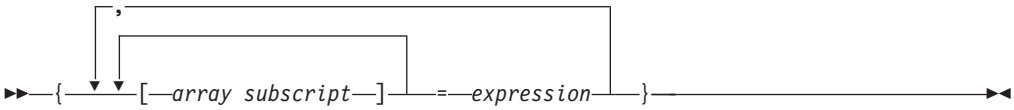
Designated initializers allow for the following flexibility:

- Elements within an aggregate can be initialized in any order.
- The initializer list can omit elements that are declared anywhere in the aggregate, rather than only at the end. Elements that are omitted are initialized as if they are static objects: arithmetic types are initialized to 0; pointers are initialized to NULL.

- Where inconsistent or incomplete bracketing of initializers for multi-dimensional arrays or nested aggregates may be difficult to understand, designators can more clearly identify the element or member to be initialized.

Designator list syntax for structures and unions

Designator list syntax for arrays



In the following example, the designator is `.any_member` and the designated initializer is `.any_member = 13`:

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

The following example shows how the second and third members `b` and `c` of structure variable `klm` are initialized with designated initializers:

```
struct xyz {
       int a;
       int b;
       int c;
       } klm = { .a = 99, .c = 100 };
```

In the following example, the third and second elements of the one-dimensional array `aa` are initialized to `3` and `6`, respectively:

```
int aa[4] = { [2] = 3, [1] = 6 };
```

The following example initializes the first four and last four elements, while omitting the middle four:

```
      static short grid[3] [4] = { [0][0]=8, [0][1]=6,
                                   [0][2]=4, [0][3]=1,
                                   [2][0]=9, [2][1]=3,
                                   [2][2]=1, [2][3]=1 };
```

The omitted four elements of `grid` are initialized to zero:

Element	Value	Element	Value
`grid[0] [0]`	8	`grid[1] [2]`	0
`grid[0] [1]`	6	`grid[1] [3]`	0
`grid[0] [2]`	4	`grid[2] [0]`	9
`grid[0] [3]`	1	`grid[2] [1]`	3
`grid[1] [0]`	0	`grid[2] [2]`	1
`grid[1] [1]`	0	`grid[2] [3]`	1

Designated initializers can be combined with regular initializers, as in the following example:

```
int a[10] = {2, 4, [8]=9, 10}
```

In this example, a[0] is initialized to 2, a[1] is initialized to 4, a[2] to a[7] are initialized to 0, and a[9] is initialized to 10.

In the following example, a single designator is used to "allocate" space from both ends of an array:

```
int a[MAX] = {
      1, 3, 5, 7, 9, [MAX-5] = 8, 6, 4, 2, 0
};
```

The designated initializer, [MAX-5] = 8, means that the array element at subscript MAX-5 should be initialized to the value 8. 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
};
```

You can also use designators to represent members of nested structures. For example:

```
struct a {
         struct b {
              int c;
              int d;
          } e;
         float f;
} g = {.e.c = 3 };
```

initializes member c of structure variable e, which is a member of structure variable g, to the value of 3.

Related reference:

"Initialization of structures and unions"

"Initialization of arrays" on page 116

Initialization of structures and unions

An initializer for a structure is a brace-enclosed comma-separated list of values, and for a union, a brace-enclosed single value. The initializer is preceded by an equal sign (=).

C99 and C++ allow the initializer for an automatic member variable of a union or structure type to be a constant or non-constant expression.

C++ The initializer for a static member variable of a union or structure type must be a constant expression or string literal. See "Static data members" on page 367 for more information. C++

There are two ways to specify initializers for structures and unions:

- With C89-style initializers, structure members must be initialized in the order declared, and only the first member of a union can be initialized.
- C Using *designated* initializers, a C99 feature which allows you to *name* members to be initialized, structure members can be initialized in any order, and

any (single) member of a union can be initialized. Designated initializers are described in detail in "Designated initializers for aggregate types (C only)" on page 110. C

Using C89-style initialization, 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"};
```

C Using a designated initializer in the same example, the following initializes the second union member `age` :

```
union {
      char birthday[9];
      int age;
      float weight;
      } people = { .age = 14 };
```

C

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"`

Unnamed structure or union members do not participate in initialization and have indeterminate value after initialization. Therefore, in the following example, the bit field is not initialized, and the initializer 3 is applied to member `b`:

```
struct {
     int a;
     int :10;
     int b;
     } w = { 2, 3 };
```

You do not have to initialize all members of structure variables. If a structure variable does not have an initializer, the initial values of the structure members depend on the storage class associated with the structure variable:

- If a structure variable has static storage, its members are implicitly initialized to zero of the appropriate type.

- If a structure variable has automatic storage, its members have no default initialization.

If a structure variable is partially initialized, all the uninitialized structure members are implicitly initialized to zero no matter what the storage class of the structure variable is. See the following example:

```
struct one {
    int a;
    int b;
    int c;
};
void main(){
    struct one z1;         // Members in z1 do not have default initial values.
    static struct one z2;  // z2.a=0, z2.b=0, and z2.c=0.
    struct one z3 = {1};   // z3.a=1, z3.b=0, and z3.c=0.
}
```

In this example, structure variable `z1` has automatic storage, and it does not have an initializer, so all the members in `z1` do not have default initial values. Structure variable `z2` has static storage, and all its members are implicitly initialized to zero. Structure variable `z3` is partially initialized, so all its uninitialized members are implicitly initialized to zero.

You do not have to initialize all members of a union. The default initializer for a union with static storage is the default for the first component. A union with automatic storage has no default initialization.

C To initialize only the third and fourth members of the `temp_address` variable, you could use a designated initializer list, as follows:

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

C

Related reference:

Structure and union variable declarations

"Explicit initialization with constructors" on page 413

"Assignment operators" on page 161

Initialization of enumerations

The initializer for an enumeration variable contains the = symbol followed by an expression *enumeration_constant*.

C++ In C++, the initializer must have the same type as the associated enumeration type. C++

The following statement declares an C++11 unscoped C++11 enumeration `grain`.

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

The following statement defines a variable g_food and initializes g_food to the value of barley. The integer value associated with barley is 2.

```
enum grain g_food = barley;
```

C++11

The following rules apply to both the scoped and unscoped enumerations.

- An enumeration cannot be initialized using an integer or enumeration constant from a different enumeration, without an explicit cast.
- An uninitialized enumeration variable has undefined value.

The following statement declares an unscoped enumeration color.

```
enum color { white, yellow, green, red, brown };
```

The following statement declares a scoped enumeration letter and references the scoped enumerators directly inside the scope of the enumeration. The initial values of A, B, C, and D are 0, 1, 1, and 2.

```
enum class letter { A, B, C = B, D = C + 1 };
```

The following statement defines a variable let1 and initializes let1 to the value of A. The integer value associated with A is 0.

```
letter let1 = letter :: A;
```

To reference scoped enumerators outside of the enumeration's scope, you must qualify the enumerators with the name of the enumeration. For example, the following statement is invalid.

```
letter let2 = A;     //invalid
```

The keyword enum in the following statement is optional and can be omitted.

```
enum letter let3 = letter :: B;
```

The white enumerator is visible in the following statement, because color is an unscoped enumeration.

```
color color1 = white;     // valid
```

Unscoped enumerations can also be qualified with their enumeration scope, for example:

```
color color2 = color :: yellow;    // valid
```

You cannot initialize an enumeration with an enumeration constant from a different enumeration or an integer without an explicit cast. For example, the following two statements are invalid.

```
letter let4 = color :: white;   // invalid

letter let5 = 1;                // invalid
```

You can use explicit cast to initialize an enumeration with an enumeration constant from a different enumeration or an integer. For example, the following two statements are valid.

```
letter let6 = (letter) color :: white;     // valid

letter let7 = (letter) 2;                  // valid
```

C++11

Related reference:

Enumeration variable declarations

Initialization of 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 time, 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.

The following examples define pointers with null pointer values:

```
char *a = 0;
char *b = NULL;
```

C++11

```
char *ch = nullptr;
```

C++11

Related reference:

"Pointers" on page 100

Initialization of 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.)

There are two ways to specify initializers for arrays:

- With C89-style initializers, array elements must be initialized in subscript order.
- C Using *designated* initializers, which allow you to specify the values of the subscript elements to be initialized, array elements can be initialized in any order. Designated initializers are described in detail in "Designated initializers for aggregate types (C only)" on page 110. C

Using C89-style initializers, 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[0]` and `number1[1]` are the same as in the previous definition, but `number1[2]` is `0`.

C

The following definition shows how you can use designated initializers to skip over elements of the array that you don't want to initialize explicitly:

```
static int number[3] = { [0] = 5, [2] = 7 };
```

The array `number` contains the following values: `number[0]` is `5`; `number[1]` is implicitly initialized to `0`; `number[2]` is `7`.

C

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.

Initialization of character arrays

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 you initialize 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. C++

Initialization of multidimensional arrays

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. 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 automatically initialized to zero.

  ```
  static short grid[3] [4] = {8, 6, 4, 1, 9, 3, 1, 1};
  ```

  The initial values of grid are:

Element	Value	Element	Value
grid[0] [0]	8	grid[1] [2]	1
grid[0] [1]	6	grid[1] [3]	1
grid[0] [2]	4	grid[2] [0]	0
grid[0] [3]	1	grid[2] [1]	0
grid[1] [0]	9	grid[2] [2]	0
grid[1] [1]	3	grid[2] [3]	0

- C Using *designated* initializers. The following example uses designated initializers to explicitly initialize only the last four elements of the array. The first eight elements that are not explicitly initialized are automatically initialized to zero.

```
static short grid[3] [4] = { [2][0] = 8, [2][1] = 6,
                             [2][2] = 4, [2][3] = 1 };
```

  The initial values of `grid` are:

Element	Value	Element	Value
grid[0] [0]	0	grid[1] [2]	0
grid[0] [1]	0	grid[1] [3]	0
grid[0] [2]	0	grid[2] [0]	8
grid[0] [3]	0	grid[2] [1]	6
grid[1] [0]	0	grid[2] [2]	4
grid[1] [1]	0	grid[2] [3]	1

C

Related reference:

"Arrays" on page 104

"Designated initializers for aggregate types (C only)" on page 110

Initialization of references (C++ only)

When you initialize a reference, you bind that reference to an object, which is not necessarily the object denoted by the initializer expression.

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 binding the reference to 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 a 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

Reference binding

Suppose `T` and `U` are two types. If ignoring top-level cv-qualifiers, `T` is of the same type as `U` or is a base class of `U`, `T` and `U` are *reference-related*.

Example 1

```
typedef int t1;
typedef const int t2;
```

In this example, `t1` and `t2` are reference-related.

If `T` and `U` are reference-related, and `T` is at least as cv-qualified as `U`, `T` is *reference-compatible* with `U`. In Example 1, `t1` is not reference-compatible with `t2`, but `t2` is reference-compatible with `t1`.

If an lvalue reference `r` to type `T` is to be initialized by an expression `e` of type `U`, and `T` is reference-compatible with `U`, the reference `r` can be bound directly to `e` or a base class subobject of `e` unless `T` is an inaccessible or ambiguous base class of `U`.

Example 2

```
int a = 1;
const int& ra = a;

struct A {};
struct B: A {} b;

A& rb = b;
```

In this example, the `const int` type is reference-compatible with the `int` type, so `ra` can be bound directly to `a`. Structure `A` is reference-related to structure `B`, so `rb` can be bound directly to `b`.

If an lvalue reference `r` to type `T` is to be initialized by an expression `e` of type `U`, `r` can be bound to the lvalue result of the conversion of `e` or a base class of `e` if the following conditions are satisfied. In this case, the conversion function is chosen by overload resolution.

- `U` is a class type.
- `T` is not reference-related to `U`.
- `e` can be converted to an lvalue of type `S`, and `T` is reference-compatible with `S`.

Example 3

```
struct A {
   operator int&();
};

const int& x= A();
```

In this example, structure `A` is a class type, and the `const int` type is not reference-related to structure `A`. However, `A` can be converted to an lvalue of type `int`, and `const int` is reference-compatible with `int`, so reference `x` of type `const int` can be bound to the conversion result of `A()`.

By default, the compiler cannot bind a non-const or volatile lvalue reference to an rvalue.

Example 4

```
int& a = 2; // error
const int& b = 1; // ok
```

In this example, the variable a is a non-const lvalue reference. The compiler cannot bind a to the temporary initialized with the rvalue expression 2, and issues an error message. The variable b is a nonvolatile const lvalue reference, which can be initialized with the temporary initialized with the rvalue expression 1.

IBM

If you specify the **LANGLVL(COMPATRVALUEBINDING)** option, the compiler can bind a non-const or volatile lvalue reference to an rvalue of a user-defined type where an initializer is not required. The default value of this option is **LANGLVL(NOCOMPATRVALUEBINDING)**. This compiler behavior conflicts with the rvalue references feature, which does not allow a non-const or volatile lvalue reference to be bound to an rvalue. If both of the features are enabled, the compiler issues an error message.

Notes:

- A non-const or volatile lvalue reference cannot be bound to an rvalue of a built-in type.
- A non-const or volatile lvalue reference that is a class member cannot be bound to an rvalue.

IBM

C++11

Suppose an expression e of type U belongs to one of the following value categories:

- An xvalue
- A class prvalue
- An array prvalue
- A function lvalue

If an rvalue reference or a nonvolatile const lvalue reference r to type T is to be initialized by the expression e, and T is reference-compatible with U, reference r can be initialized by expression e and bound directly to e or a base class subobject of e unless T is an inaccessible or ambiguous base class of U.

Example 5

```
int& func1();
int& (&&rf1)()=func1;

int&& func2();
int&& rf2 = func2();

struct A{
   int arr[5];
};
int(&&ar_ref)[5] = A().arr;
A&& a_ref = A();
```

In this example, rf1, rf2, ar_ref, and a_ref are all rvalue references. rf1 is bound to the function lvalue func1, rf2 is bound to the xvalue result of the call func2(), ar_ref is bound to the array prvalue A().arr, and a_ref is bound to the class prvalue A().

Suppose r is an rvalue reference or nonvolatile const lvalue reference to type T, and r is to be initialized by an expression e of type U. r can be bound to the conversion result of e or a base class of e if the following conditions are satisfied. In this case, the conversion function is chosen by overload resolution.

- U is a class type.
- T is not reference-related to U.
- e can be converted to an xvalue, class prvalue, or function lvalue type of S, and T is reference-compatible with S.

Example 6

```
int i;
struct A {
   operator int&&() {
     return static_cast<int&&>(i);
   }
 };

const int& x = A();

int main() {
   assert(&x == &i);
}
```

In this example, structure A is a class type, and the const int type is not reference-related to structure A. However, A can be converted to an xvalue of type int, and const int is reference-compatible with int, so reference x of const int can be initialized with A() and bound to variable i.

An rvalue reference can be initialized with an lvalue in the following contexts:

- A function lvalue
- A temporary converted from an lvalue
- An rvalue result of a conversion function for an lvalue object that is of a class type

Example 7

```
int i = 1;
int&& a = 2; // ok
int&& b = i; // error
double&& c = i; // ok
```

In this example, the rvalue reference a can be bound to the temporary initialized with the rvalue expression 2, but the rvalue reference b cannot be bound to the lvalue expression i. You can bind the rvalue reference c to the temporary value 1.0 that is converted from the variable i.

C++11

Related reference:

"References (C++ only)" on page 107

"Pass by reference (C++ only)" on page 252

"Lvalues and rvalues" on page 141

Initialization of complex types (C11)

When the C11 complex initialization feature is enabled, you can initialize C99 complex types with a value of the form `x + yi`, where `x` and `y` can be any floating point value, including `Inf` or `NaN`.

C The C11 complex initialization feature can be enabled by the **-qlanglvl=extc1x** group option. C

C++ The C11 complex initialization feature can be enabled by the **-qlanglvl=extended** or **-qlanglvl=extended0x** group option. You can also use the **-qlanglvl=complexinit** suboption to enable this feature. When you specify the **-qlanglvl=nocomplexinit** option, only the C11 form of complex initialization is disabled. C++

C To enable the initialization of these complex types, macros **CMPLX**, **CMPLXF**, and **CMPLXL** are defined inside the standard header file `complex.h` for C11 compilation, which act as if the following functions are used.

```
float complex CMPLXF( float x, float y );
double complex CMPLX( double x, double y );
long double complex CMPLXL( long double x, long double y );
```

Note: These macros might infringe upon user namespaces. You must avoid using the macro names for other purposes.
C

These macros are available only if the C language header file `complex.h` is included, and they result in values that are suitable for static initialization if arguments are suitable for static initialization. C++ To use the C language header file `complex.h` in C++ programs, you must specify the **-qlanglvl=c99complexheader** or **-qlanglvl=c99complex** option. C++

The following example shows how to initialize a complex type with a value of the form *x + yi*.

```
// a.c
#include <stdio.h>
#include <math.h>
#include <complex.h>

int main(void) {
  float _Complex c = CMPLXF(5.0, NAN);
  printf("Value: %e + %e * I\n", __real__(c), __imag__(c));
  return;
}
```

You can specify the following command to compile this program:

```
xlc -qlanglvl=extc1x -qfloat=ieee a.c
```

The result of running the program is:

```
Value: 5 + NaNQ * I
```

Related reference:

"Extensions for C11 compatibility" on page 640

"Floating-point literals" on page 23

"Floating-point types" on page 56

Declarator qualifiers

z/OS XL C/C++ includes two additional qualifiers that are given in declarator specifications:

- C _Packed C
- C++ _Export C++

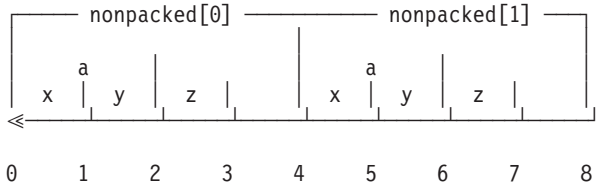
The _Packed qualifier (C only)

The z/OS XL C compiler aligns structure and union members according to their natural byte boundaries and ends the structure or union on its natural boundary. However, since the alignment of a structure or union is that of the member with the largest alignment requirement, the compiler may add padding to elements whose byte boundaries are smaller than this requirement. You can use the `_Packed` qualifier to remove padding between members of structures or unions. Packed and nonpacked structures and unions have different storage layouts.

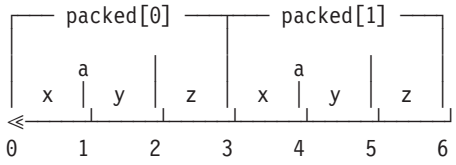
Consider the following example:

```
union uu{
   short    a;
   struct {
      char x;
      char y;
      char z;
   } b;
};

union uu           nonpacked[2];
_Packed union uu   packed[2];
```

In the array of unions `nonpacked`, since the largest alignment requirement among the union members is that of `short a`, namely, 2 bytes, one byte of padding is added at the end of each union in the array to enforce this requirement:



In the array of unions `packed`, each union has a length of only 3 bytes, as opposed to the 4 bytes of the previous case:



Note: When the `_Packed` qualifier is used, the compiler removes padding between members of structures or unions, regardless of the member type.

If you specify the `_Packed` qualifier on a structure or union that contains a structure or union as a member, the qualifier is not passed on to the nested structure or union.

Related reference:

"Compatibility of structures, unions, and enumerations (C only)" on page 73

"#pragma pack" on page 583

The _Export qualifier (C++ only)

You can use the `_Export` keyword with a function name or external variable to declare that it is to be exported (made available to other modules). The `_Export` keyword must immediately precede the object name. For more information, see "The _Export function specifier (C++ only)" on page 233.

Related reference:

"External linkage" on page 8

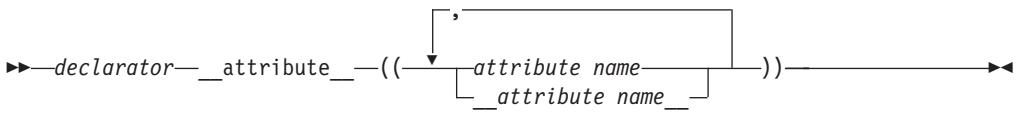
"#pragma export" on page 553

Variable attributes (IBM extension)

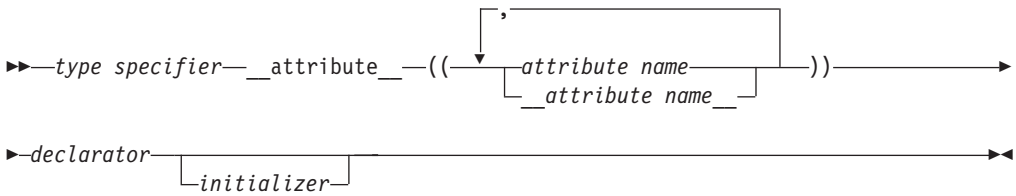
A variable attribute is a language extension that allows you to use a named attribute to specify special properties of variables. Currently, only the variable attribute `aligned` is supported on the z/OS platform.

A variable attribute is specified with the keyword `__attribute__` followed by the attribute name and any additional arguments the attribute name requires. A variable `__attribute__` specification is included in the declaration of a variable, and can be placed before or after the declarator. Although there are variations, the syntax generally takes either of the following forms:

Variable attribute syntax: post-declarator



Variable attribute syntax: pre-declarator



You can specify *attribute name* with or without leading and trailing double underscore characters; however, using the double underscore characters reduces the likelihood of name conflicts with macros of the same name. For unsupported attribute names, the z/OS XL C/C++ compiler issues diagnostics and ignores the attribute specification. Multiple attribute names can be specified in the same attribute specification.

In a comma-separated list of declarators on a single declaration line, if a variable attribute appears before all the declarators, it applies to all declarators in the

declaration. If the attribute appears after a declarator, it only applies to the immediately preceding declarator. For example:

```
struct A  {

 int b __attribute__((aligned));          /* typical placement of variable */
                                                      /* attribute */

 int __attribute__((aligned))__ c = 10;   /* variable attribute can also be */
                                            /* placed here */

 int d, e, f __attribute__((aligned));    /* attribute applies to f only */

 int g __attribute__((aligned)), h, i;    /* attribute applies to g only */

 int __attribute__((aligned)) j, k, l;    /* attribute applies to j, k, and l */

 };
```

The following variable attributes are supported:

- "The aligned variable attribute"

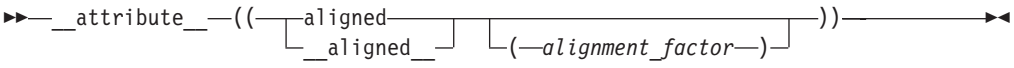
The aligned variable attribute

With the `aligned` variable attribute, you can override the default memory alignment mode to specify a minimum memory alignment value, expressed as a number of bytes, for any of the following types of variables:

- Non-aggregate variables
- Aggregate variables (such as a structures, classes, or unions)
- Selected member variables

The attribute is typically used to increase the alignment of the given variable.

aligned variable attribute syntax



The *alignment_factor* is the number of bytes, specified as a constant expression that evaluates to a positive power of 2. On the z/OS platform, the maximum supported value is 8 bytes in 32-bit mode, and 16 bytes in 64-bit mode. If you omit the alignment factor (and its enclosing parentheses), the compiler automatically uses the platform maximum. If you specify an alignment factor greater than the maximum, the compiler uses the default alignment in effect and ignores your specification.

When you apply the `aligned` attribute to a member variable in a bit field structure, the attribute specification is applied to the bit field container. If the default alignment of the container is greater than the alignment factor, the default alignment is used.

Example

In the following example, the structures `first_address` and `second_address` are set to an alignment of 16 bytes:

```
struct address {
                int street_no;
                char *street_name;
                char *city;
                char *prov;
                char *postal_code;
```

```
              } first_address __attribute__((__aligned__(16))) ;

struct address second_address __attribute__((__aligned__(16))) ;
```

In the following example, only the members `first_address.prov` and `first_address.postal_code` are set to an alignment of 16 bytes:

```
struct address {
                int street_no;
                char *street_name;
                char *city;
                char *prov __attribute__((__aligned__(16))) ;
                char *postal_code __attribute__((__aligned__(16))) ;
              } first_address  ;
```

Chapter 5. Type conversions

An expression of a given type is *implicitly converted* when it is used in the following situations:

- As an operand of an arithmetic or logical operation.
- As a condition in an `if` statement or an iteration statement (such as a `for` loop). The expression will be converted to a Boolean (or an integer in C89).
- In a `switch` statement. The expression is converted to an integral type.
- As the right operand of an assignment or as an initializer.
- As an initialization. This includes the following types:
  - 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.

C The implicit conversion result is an rvalue. C

C++ The implicit conversion result belongs to one of the following value categories depending on different converted expressions types:

- An lvalue if the type is an lvalue reference type C++11 or an rvalue reference to a function type C++11
- C++11 An xvalue if the type is an rvalue reference to an object type C++11
- A C++11 (prvalue) C++11 rvalue in other cases

C++

You can perform *explicit* type conversions using a *cast* expression, as described in "Cast expressions" on page 176.

Related reference:

"User-defined conversions" on page 423

"Conversion constructors" on page 424

"Conversion functions" on page 426

"The switch statement" on page 202

"The if statement" on page 200

"The return statement" on page 212

"Lvalues and rvalues" on page 141

"References (C++ only)" on page 107

Arithmetic conversions and promotions

The following sections discuss the rules for the standard conversions for arithmetic types:

If two operands in an expression have different types, they are subject to the rules of the *usual arithmetic conversions*, as described in "Usual arithmetic conversions" on page 133.

Integral conversions

Unsigned integer to unsigned integer or signed integer to signed integer
If the types are identical, there is no change. If the types are of a different size, and the value can be represented by the new type, the value is not changed; if the value cannot be represented by the new type, truncation or sign shifting will occur.

Signed integer to unsigned integer
The resulting value is the smallest unsigned integer type congruent to the source integer. If the value cannot be represented by the new type, truncation or sign shifting will occur.

Unsigned integer to signed integer
If the signed type is large enough to hold the original value, there is no change. If the value can be represented by the new type, the value is not changed; if the value cannot be represented by the new type, truncation or sign shifting will occur.

Signed and unsigned character types to integer
The character types are promoted to type `int`.

Wide character type wchar_t to integer
If the original value can be represented by `int`, it is represented as `int`. If the value cannot be represented by `int`, it is promoted to the smallest type that can hold it: `unsigned int`, `long`, or `unsigned long`.

Signed and unsigned integer bit field to integer
If the original value can be represented by `int`, it is represented as `int`. If The value cannot be represented by `int`, it is promoted to `unsigned int`.

Enumeration type to integer
If the original value can be represented by `int`, it is represented as `int`. If the value cannot be represented by `int`, it is promoted to the smallest type that can hold it: `unsigned int`, `long`, or `unsigned long`. Note that an enumerated type can be converted to an integral type, but an integral type cannot be converted to an enumeration.

Boolean conversions

An unscoped enumeration, pointer, or pointer to member type can be converted to a Boolean type.

C If the scalar value is equal to 0, the Boolean value is 0; otherwise, the Boolean value is 1. C

C++ A zero, null pointer, or null member pointer value is converted to `false`. All other values are converted to `true`. C++

C++11 A null pointer with the `nullptr` value is converted to `false`. C++11

Floating-point conversions

The standard rule for converting between real floating-point types is as follows:

If the value being converted can be represented exactly in the new type, it is unchanged. If the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is rounded, according to the current compile-time or runtime rounding mode in effect. If the value being converted is outside the range of values that can be represented, the result is dependent on the rounding mode.

Integer to floating point
If the value being converted can be represented exactly in the new type, it is unchanged. If the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is correctly rounded. If the value being converted is outside the range of values that can be represented, the result is quiet NaN.

Floating point to integer
The fractional part is discarded (i.e., the value is truncated toward zero). If the value of the integral part cannot be represented by the integer type, the result is one of the following:

- If the integer type is unsigned, the result is the largest representable number if the floating-point number is positive, or 0 otherwise.
- If the integer type is signed, the result is the most negative or positive representable number according to the sign of the floating-point number.

C11 **Note:** The conversion between a floating type and a pointer type is not allowed. C11

Implicit conversions of decimal floating-point types (IBM extension)

The compiler has the following decimal floating-point types:

- `_Decimal32`
- `_Decimal64`
- `_Decimal128`

The following implicit conversions are always supported:

- Implicit conversions between decimal floating-point types:
  - `_Decimal32` to `_Decimal64`
  - `_Decimal32` to `_Decimal128`
  - `_Decimal64` to `_Decimal32`
  - `_Decimal64` to `_Decimal128`
  - `_Decimal128` to `_Decimal32`
  - `_Decimal128` to `_Decimal64`
- Implicit conversions between decimal floating-point types and the following integer types:
  - `signed char, unsigned char`
  - `signed short int, unsigned short int`
  - `signed int, unsigned int`
  - `signed long int, unsigned long int`
  - `signed long long int, unsigned long long int`
- Implicit conversions between decimal floating-point types and Boolean types `bool` or `_Bool`.

Implicit conversions between decimal floating-point types and the following generic floating-point types are supported conditionally. It is supported through assignment operation using the simple assignment operator =, initialization, function argument passing and function return statements.

- `float`
- `double`
- `long double`

The following examples demonstrate the implicit conversion from a generic floating-point type to a decimal floating-point type. In this example, variable `f1` is implicitly converted from type `float` to type `_Decimal32` in the initialization.

```
float f1;
_Decimal32 d1 = f1;
```

C++

```
float f1;
_Decimal32 d1(f1);
```

C++

Restriction: You cannot mix decimal floating-point types with generic floating-point types or complex floating-point types in arithmetic expressions unless you use explicit conversions. Here is an example:

```
_Decimal32 d1;
float f1;
float f2 = f1 + d1;         // Incorrect
float f3 = f1 + (float)d1;  // Correct
```

Complex conversions

Complex to complex
If the types are identical, there is no change. If the types are of a different size, and the value can be represented by the new type, the value is not changed; if the value cannot be represented by the new type, both real and imaginary parts are converted according to the standard conversion rule given above.

Complex to real (binary)
The imaginary part of the complex value is discarded. If necessary, the value of the real part is converted according to the standard conversion rule given above.

Real (binary) to complex
The source value is used as the real part of the complex value, and converted, if necessary, according to the standard conversion rule given above. The value of the imaginary part is zero.

Related reference:

"Floating-point types" on page 56

Packed decimal conversions (C only)

Packed decimal to long long integer
The fractional part is discarded.

Long long integer to packed decimal
The resulting size is `decimal(20,0)`.

Complex to packed decimal
Only the floating value of the real part is used.

Packed decimal to complex
The real part of the complex type is converted, and the imaginary part is 0.

Packed decimal to decimal floating-point
If the number of significant digits in the packed decimal value exceeds the precision of the target, the result is rounded to the target precision using the current decimal floating-point rounding mode.

Decimal floating-point to packed decimal
Before conversion, the decimal floating-point value is rounded or truncated to match the fractional precision of the resulting type, if necessary. If the value being converted represents infinity or NaN, or if non-zero digits are truncated from the left end of the result, the result is undefined.

Usual arithmetic conversions

When different arithmetic types are used as operands in certain types of expressions, standard conversions known as *usual arithmetic conversions* are applied.

For example, when the values of two different integral types are added together, 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. Chapter 6, “Expressions and operators,” on page 141 provides a list of the operators and expressions that participate in the usual arithmetic conversions.

Conversion ranks for arithmetic types

The ranks in the tables are listed from highest to lowest:

Table 20. Conversion ranks for floating-point types

Operand type
`long double` or `long double _Complex`
`double` or `double _Complex`
`float` or `float _Complex`

IBM

Table 21. Conversion ranks for decimal floating-point types

Operand type
_Decimal128
_Decimal64
_Decimal32

IBM

Table 22. Conversion ranks for integer types

Operand type
`long long int, unsigned long long int`

Table 22. Conversion ranks for integer types (continued)

Operand type
`long int, unsigned long int`
`int, unsigned int`
`short int, unsigned short int`
`char, signed char, unsigned char`
Boolean
Notes: • The `long long int` and `unsigned long long int` types are not included in the C89, C++98 and C++03 standards. • C The wchar_t type is not a distinct type, but rather a typedef for an integer type. The rank of the wchar_t type is equal to the rank of its underlying type. C • C The rank of enumerated type is equal to the rank of its underlying type. C

Rules for floating-point operands

In a context where an operation involves two operands, if either of the operands is of floating-point type, the compiler performs the usual arithmetic conversions to bring these two operands to a common type. The floating-point promotions are applied to both operands. The following rules apply to the promoted operands:

1. If both operands have the same type, no conversion is needed.
2. Otherwise, if both operands have complex types, the type at a lower integer conversion rank is converted to the type at a higher rank. For more information, see "Floating-point conversions" on page 130.
3. Otherwise, if one operand has a complex type, the type of both operands after conversion is the higher rank of the following types:
   - The complex type corresponding to the type of the generic floating-point operand
   - The type of the complex operand

   For more information, see "Floating-point conversions" on page 130.
4. Otherwise, both operands have generic floating types. The following rules apply:
   a. If one operand has the `long double` type, the other operand is converted to `long double`.
   b. Otherwise, if one operand has the `double` type, the other operand is converted to `double`.
   c. Otherwise, if one operand has the `float` type, the other operand is converted to `float`.

Rules for integral operands

In a context where an operation involves two operands, if both of the operands are of integral types, the compiler performs the usual arithmetic conversions to bring these two operands to a common type. The integral promotions are applied to both operands and the following rules apply to the promoted operands:

1. If both operands have the same type, no conversion is needed.

2. Otherwise, if both operands have signed integer types or both have unsigned integer types, the type at a lower integer conversion rank is converted to the type at a higher rank.
3. Otherwise, if one operand has an unsigned integer type and the other operand has a signed integer type, the following rules apply:
   a. If the rank for the unsigned integer type is higher than or equal to the rank for the signed integer type, the signed integer type is converted to the unsigned integer type.
   b. Otherwise, if the signed integer type can represent all of the values of the unsigned integer type, the unsigned integer type is converted to the signed integer type.
   c. Otherwise, both types are converted to the unsigned integer type that corresponds to the signed integer type.

Related reference:

"Integral types" on page 54

"Boolean types" on page 55

"Floating-point types" on page 56

"Character types" on page 59

"Enumerations" on page 70

"Binary expressions" on page 160

Integral and floating-point promotions

The *integral and floating-point promotions* are used automatically as part of the usual arithmetic conversions and default argument promotions. The integral and floating-point promotions do not change either the sign or the magnitude of the value. For more information about the usual arithmetic conversions, see "Usual arithmetic conversions" on page 133.

C++11

Integral promotion rules for wchar_t

If a value is of the wchar_t type, the type of the value can be converted to the first of the following types that can represent all the values of the underlying type of wchar_t:

- `int`
- `unsigned int`
- `long int`
- `unsigned long int`
- `long long int`
- `unsigned long long int`

If none of the types in the list can represent all the values of the underlying type of wchar_t, the wchar_t type is converted to the underlying type of wchar_t.

C++11

Integral promotion rules for bit field

C The rules apply to the following conditions:

- The **-qupconv** option is in effect.
- The type of an integral bit field is `unsigned`.
- The type of the integral bit field is smaller than the `int` type.

If all these conditions are satisfied, one of the following rules applies to the promotion of the integral bit field:

- If the `unsigned int` type can represent all the values of the integral bit field, the bit field is converted to `unsigned int`.
- Otherwise, no integral promotion applies to the bit field.

If any of these conditions is not satisfied, one of the following rules applies to the promotion of the integral bit field:

- If the `int` type can represent all the values of the integral bit field, the bit field is converted to `int`.
- Otherwise, if the `unsigned int` type can represent all the values, the bit field is converted to `unsigned int`.
- Otherwise, no integral promotion applies to the bit field.

C

C++ One of the following rules applies to an integral bit field promotion:

- If the `int` type can represent all the values of an integral bit field, the bit field is converted to `int`.
- Otherwise, if the `unsigned int` type can represent all the values, the bit field is converted to `unsigned int`.
- Otherwise, no integral promotion applies to the bit field.

C++

Integral promotion rules for Boolean

C If the **-qupconv** option is in effect, a Boolean value is converted to the `unsigned int` type with its value unchanged. Otherwise, if the **-qnoupconv** option is in effect, a Boolean value is converted to the `int` type with its value unchanged.

C

C++ If a Boolean value is `false`, it is converted to an `int` with a value of 0. If a Boolean value is `true`, it is converted to an `int` with a value of 1.

Integral promotion rules for other types

C The rules apply to the following conditions:

- The **-qupconv** option is in effect.
- The type of an integer type other than bit field and Boolean is `unsigned`.
- The type of the integer type is smaller than the `int` type.

If all these conditions are satisfied, the integer type is converted to the `unsigned int` type.

If any of these conditions is not satisfied, one of the following rules applies to the promotion of the integer type:

- If the integer type can be represented by the `int` type and its rank is lower than the rank of `int`, the integer type is converted to the `int` type.
- Otherwise, the integer type is converted to the `unsigned int` type.

C

C++ One of the following rules applies to the promotion of an integer type other than wchar_t, bit field, and Boolean:

- If the integer type can be represented by the `int` type and its rank is lower than the rank of `int`, the integer type is converted to the `int` type.
- Otherwise, the integer type is converted to the `unsigned int` type.

C++

floating-point promotion rules

The `float` type can be converted to the `double` type. The `float` value is not changed after the promotion.

Lvalue-to-rvalue conversions

If an lvalue C++11 or xvalue C++11 is used in a situation in which the compiler expects a C++11 (prvalue) C++11 rvalue, the compiler converts the lvalue C++11 or xvalue C++11 to a C++11 (prvalue) C++11 rvalue. However, a C++11 (prvalue) C++11 rvalue cannot be converted implicitly to an lvalue C++11 or xvalue C++11 , C++ except by user-defined conversions C++ . The following table lists exceptions to this rule.

Situation before conversion	Resulting behavior
The lvalue is a function type.	A pointer to function
The lvalue is an array.	A pointer to the first element of the array
The type of the lvalue C++11 or xvalue C++11 is an incomplete type.	compile-time error
The lvalue C++11 or xvalue C++11 refers to an uninitialized object.	undefined behavior
The lvalue C++11 or xvalue C++11 refers to an object not of the type of the C++11 (prvalue) C++11 rvalue, nor of a type derived from the type of the C++11 (prvalue) C++11 rvalue.	undefined behavior
C++ The lvalue C++11 or xvalue C++11 is a nonclass type, qualified by either `const` or `volatile`. C++	The type after conversion is not qualified by either `const` or `volatile`.

Related reference:

"Lvalues and rvalues" on page 141

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 23. 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 or more cv-qualifiers as compared to those of the right operand.

C

Zero constant to null pointer

An integral constant expression that evaluates to zero is a *null pointer constant*. This expression can be converted to a pointer. This pointer is a null pointer (pointer with a zero value), and is guaranteed not to point to any object.

C++ A constant expression that evaluates to zero can also be converted to the null pointer to a member.

Array to pointer

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. This conversion is not performed if the expression is used as the operand of the address operator `&` or the `sizeof` operator C++ or when the array is bound to a reference of the array type C++ .

Function to pointer

An lvalue that is a function can be converted to a C++11 (prvalue) C++11 rvalue that is a pointer to a function of the same type, except when the expression is used as the operand of the `&` (address) operator, the `()` (function call) operator, or the `sizeof` operator.

C11 **Note:** The conversion between a floating type and a pointer type is not allowed. C11

Related reference:

"Pointers" on page 100

"Integer constant expressions" on page 144

"Arrays" on page 104

"Pointers to functions" on page 257

"Pointers to members" on page 362

Ambiguous base classes (C++ only)

"Lvalues and rvalues" on page 141

Conversion to 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.

Any pointer to an object, 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 24. Legal assignment conversions in C for void**

Left operand type	Permitted right operand types
`(void*)`	• The constant 0. • A pointer to an object. The object may be of incomplete type. • `(void*)`

C

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. C++

Related reference:

"The void type" on page 59

Reference conversions (C++ only)

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.

Related reference:

"References (C++ only)" on page 107

"Initialization of references (C++ only)" on page 119

"Function calls" on page 249

"Function return values" on page 235

Function argument conversions

When a function is called, if a function declaration is present and includes declared argument types, the compiler performs type checking. 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;
}
```

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:

- The integral and floating-point promotions are performed.
- Arrays or functions are converted to pointers.

Related reference:

"Integral and floating-point promotions" on page 135

"Function call expressions" on page 149

"Function calls" on page 249

Chapter 6. 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.

"Operator precedence and associativity" on page 190 provides tables listing the precedence of all the operators described in the various sections listed above.

C++ C++ operators can be defined to behave differently when applied to operands of class type. This is called operator *overloading*, and is described in "Overloading operators" on page 329. C++

Lvalues and rvalues

Expressions can be categorized into one of the following value categories:

Lvalue
An expression can appear on the left side of an assignment expression if the expression is not `const` qualified.

C++11 **Xvalue** C++11
An rvalue reference that is to expire.

C++11 **(Prvalue)** C++11 **rvalue**
A C++11 non-xvalue C++11 expression that appears only on the right side of an assignment expression.

C++11 Rvalues include both xvalues and prvalues. Lvalues and xvalues can be referred as glvalues. C++11

Notes:

- Class C++11 (prvalue) C++11 rvalues can be cv-qualified, but non-class C++11 (prvalue) C++11 rvalues cannot be cv-qualified.
- Lvalues C++11 and xvalues C++11 can be of incomplete types, but C++11 (prvalue) C++11 rvalues must be of complete types or `void` types.

An object is a region of storage that can be examined and stored into. An lvalue C++11 or xvalue C++11 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. Lvalues of the following object types are not modifiable lvalues:

- An array type
- An incomplete type
- A `const`-qualified type
- A structure or union type with one of its members qualified as a `const` type

Because these lvalues are not modifiable, they cannot appear on the left side of an assignment statement, C++ except where a suitable assignment operator exists C++ .

C 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

C++ A function call that returns an lvalue reference is an lvalue. Expressions can produce an lvalue, C++11 an xvalue C++11 , a C++11 (prvalue) C++11 rvalue, or no value. C++

Certain C++ built-in C++ operators require lvalues for some of their operands. The following table lists these operators and additional constraints on their usage.

Operator	Requirement
& (unary)	Operand must be an lvalue.
++ --	Operand must be a modifiable lvalue. This applies to both prefix and postfix forms.
= += -= *= %= <<= >>= &= ^= |=	Left operand must be a modifiable 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.

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`
`*ptr = newvalue`	`*ptr`
`a++`	`a`
C++ `f()`	The function call to `f()`

C++11

The following expressions are xvalues:

- The result of calling a function whose return type is of an rvalue reference type
- A cast to an rvalue reference
- A nonstatic data member of a non-reference type accessed through an xvalue expression
- A pointer to member access expression in which the first operand is an xvalue expression and the second operand is of a pointer to member type

See the following example:

```
int a;
int&& b= static_cast<int&&>(a);

struct str{
```

```
   int c;
};

int&& f(){
   int&& var =1;
   return var;
}

str&& g();
int&& rc = g().c;
```

In this example, The initializer for rvalue reference `b` is an xvalue because it is a result of a cast to an rvalue reference. A call to the function `f()` produces an xvalue because the return type of this function is of the `int&&` type. The initializer for rvalue reference `rc` is an xvalue because it is an expression that accesses a nonstatic non-reference data member `c` through an xvalue expression.

C++11

Related reference:

"Arrays" on page 104

"Lvalue-to-rvalue conversions" on page 137

"References (C++ only)" on page 107

Primary expressions

Primary expressions fall into the following general categories:

- Names (identifiers)
- Literals (constants)
- Integer constant expressions
- Identifier expressions
- Parenthesized expressions ( )
- C11 Generic selection C11
- C++ The `this` pointer (described in "The this pointer" on page 363)
- Names qualified by the scope resolution operator (`::`) C++

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.

Table 25. 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	yes
Enumeration constant	The associated integer value	no
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 no C C++ yes C++

Table 25. Primary expressions: Names (continued)

Name declared as	Evaluates to	Is 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 no C++ yes

As an expression, a name may not refer to a label, `typedef` name, structure member, union member, structure tag, union tag, or enumeration tag. Names used for these purposes reside in a namespace that is separate from that of names used in expressions. However, 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 members; `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 reference:

"Literals" on page 19

"The this pointer" on page 363

Integer constant expressions

An *integer constant* is a value that is determined at compile time and cannot be changed at run time. An *integer constant expression* is an expression that is composed of constants and evaluated to a constant at compile time.

An integer constant expression is an expression that is composed of only the following elements:

- literals
- enumerators
- `const` variables initialized with compile-time constant expressions or C++11 `constexpr` expressions C++11
- `static const` data members initialized with compile-time constant expressions or C++11 `constexpr` expressions C++11
- casts to integral types
- `sizeof` expressions, where the operand is not a variable length array

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 enumeration 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.)

Note: C++11 The C++11 standard generalizes the concept of constant expressions. For more information, see "Generalized constant expressions (C++11)" on page 149. C++11

Related reference:

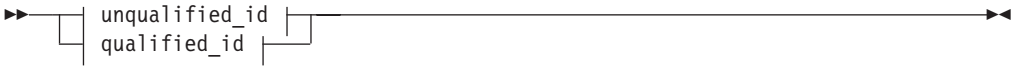
"The sizeof operator" on page 157

Identifier expressions (C++ only)

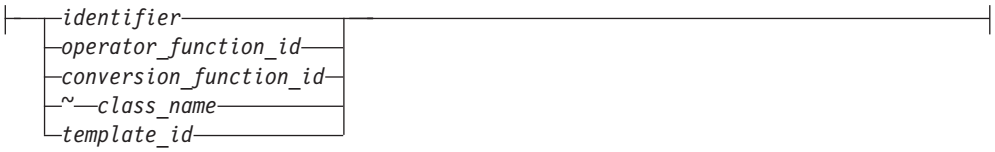
An identifier expression, or *id-expression*, is a restricted form of primary expression. Syntactically, an *id-expression* requires a higher level of complexity than a simple identifier to provide a name for all of the language elements of C++.

An *id-expression* can be either a qualified or unqualified identifier. It can also appear after the dot and arrow operators.

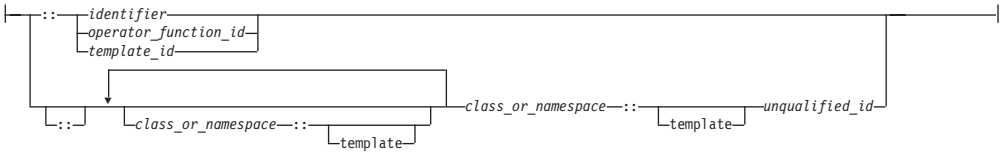
Identifier expression syntax



unqualified_id:



qualified_id:



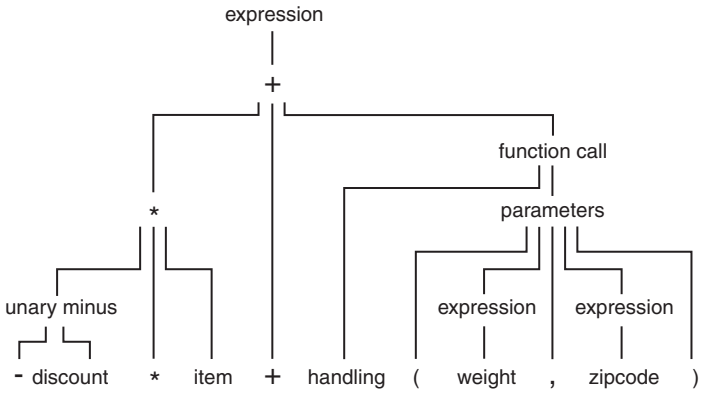
Related reference:

"Identifiers" on page 16

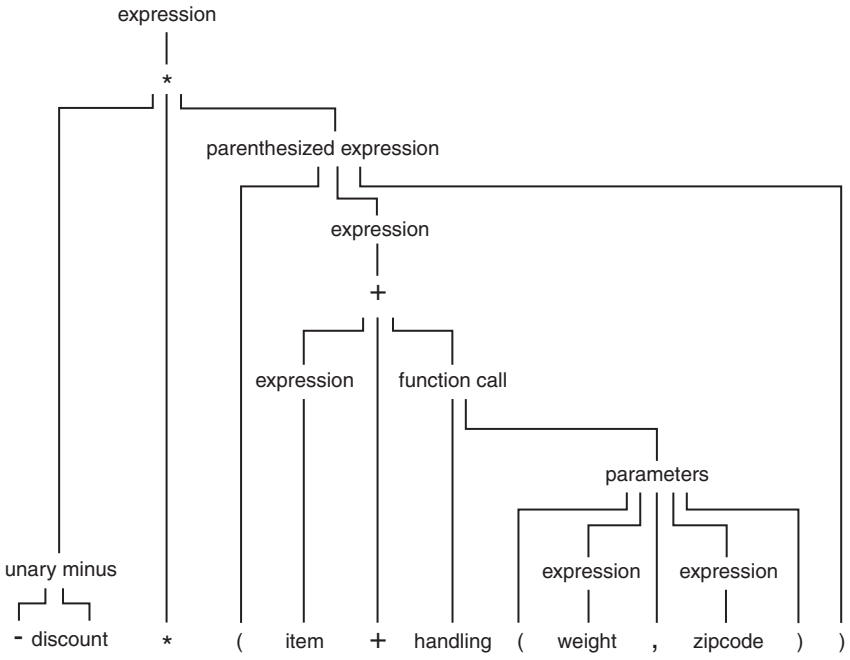
Chapter 4, "Declarators," on page 97

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);
```

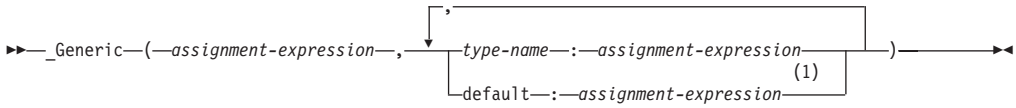
Related reference:

"Operator precedence and associativity" on page 190

Generic selection (C11)

A generic selection is a primary expression. Its type and value depend on the selected generic association.

The following diagram shows the generic selection syntax:



Notes:

1 A generic selection can have at most one default generic association.

where:

type-name
Specifies the type of a generic association. The type name that you specify in a generic association must be a complete object type other than a variably modified type.

assignment-expression
Is an assignment expression. The first assignment expression is called the controlling expression.

The generic association list is a group of generic associations. There are two forms of generic associations:

- `type-name`: assignment-expression
- `default`: assignment-expression

One generic selection cannot have two or more generic associations that specify compatible types. In one generic selection, the controlling expression can have at most one compatible type name in the generic association list. If a generic selection has no default generic association, its controlling expression must have exactly one compatible type name in its generic association list.

If there is a generic association with a type name that is compatible with the controlling expression in the generic selection, the expression in the generic selection is the result expression. Otherwise, the result expression of the generic selection is the expression in the default generic association. The controlling expression of a generic selection is not evaluated. None of the expressions from any other generic association of the generic selection is evaluated.

The type and value of a generic selection are identical to those of its result expression. For example, a generic selection is an lvalue, a function designator, or a void expression if its result expression is an lvalue, a function designator, or a void expression.

Example

The following sample `myprogram.c` defines a type-generic macro:

```
#define myfunction(X) _Generic((X), \
long double:myfunction_longdouble, \
default:myfunction_double, \
float:myfunction_float \
)(X)
void myfunction_longdouble(long double x){printf("calling %s\n",__func__);}
void myfunction_double(double x){printf("calling %s\n",__func__);}
void myfunction_float(float x){printf("calling %s\n",__func__);}

int main()
{
  long double ld;
  double d;
```

```
  float f;
  myfunction(ld);
  myfunction(d);
  myfunction(f);
}
```

When you execute the program:

```
xlc myprogram.c  -qlanglvl=extc1x
./a.out
```

the result is as follows:

```
calling myfunction_longdouble
calling myfunction_double
calling myfunction_float
```

Scope resolution operator :: (C++ only)

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 reference:

"Scope of class names" on page 351

Chapter 9, "Namespaces (C++ only)," on page 317

Generalized constant expressions (C++11)

The C++11 standard generalizes the concept of constant expressions and introduces a new keyword `constexpr` as a declaration specifier. A constant expression is an expression that can be evaluated at compile time by the compiler. The major benefits of this feature are as follows:

- Improves type safety and portability of code that requires compile-time evaluation
- Improves support for systems programming, library building, and generic programming
- Improves the usability of Standard Library components. Library functions that can be evaluated at compile time can be used in contexts that require constant expressions.

An object declaration with the `constexpr` specifier declares that object to be constant. The `constexpr` specifier can be applied only to the following contexts:

- The definition of an object
- The declaration of a function or function template
- The declaration of a static data member of a literal type

If you declare a function that is not a constructor with a `constexpr` specifier, then that function is a `constexpr` function. Similarly, if you declare a constructor with a `constexpr` specifier, then that constructor is a `constexpr` constructor.

With this feature, constant expressions can include calls to template and non-template `constexpr` functions, `constexpr` objects of class literal types, and references bound to `const` objects that are initialized with constant expressions.

Evaluations of floating-point operations at compile time use the default semantics of the **FLOAT** option.

Related reference:

"The constexpr specifier (C++11)" on page 83

"Constexpr functions (C++11)" on page 313

"Constexpr constructors (C++11)" on page 411

"C++11 compatibility" on page 640

Function call expressions

A *function call* is an expression containing the function name followed by the function call operator, `()`. 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. The argument list can contain any number of expressions separated by commas. The argument list can also be empty.

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`.

C A function call is always an rvalue. C

C++ A function call belongs to one of the following value categories depending on the result type of the function:

- An lvalue if the result type is an lvalue reference type C++11 or an rvalue reference to a function type C++11

- C++11 An xvalue if the result type is an rvalue reference to an object type C++11
- A C++11 (prvalue) C++11 rvalue in other cases

C++

Here are some examples of the function call operator:

```
stub()
overdue(account, date, amount)
notify(name, date + 5)
report(error, time, date, ++num)
```

The order of evaluation for function call arguments is not specified. In the following example:

```
method(sample1, batch.process--, batch.process);
```

the argument `batch.process--` might be evaluated last, causing the last two arguments to be passed with the same value.

Related reference:

"Function argument conversions" on page 139

"Function calls" on page 249

"Lvalues and rvalues" on page 141

"References (C++ only)" on page 107

Member expressions

Member expressions indicate members of classes, structures, or unions. The member operators are:

- Dot operator .
- Arrow operator ->

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. (A *pseudo-destructor* is a destructor of a nonclass type.) 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.

Related reference:

Access to structure and union members

"Pseudo-destructors" on page 422

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 reference:

"Pointers" on page 100

Access to structure and union members

"Structures and unions" on page 60

Chapter 12, "Class members and friends (C++ only)," on page 357

"Pseudo-destructors" on page 422

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, as shown in Table 29 on page 191.

As indicated in the descriptions of the operators, the usual arithmetic conversions are performed on the operands of most unary expressions.

The supported unary operators are:

- "Increment operator ++"
- "Decrement operator --" on page 152
- "Unary plus operator +" on page 153
- "Unary minus operator -" on page 153
- "Logical negation operator !" on page 153
- "Bitwise negation operator ~" on page 153
- "Address operator &" on page 154
- "Indirection operator *" on page 155
- C++ typeid C++
- IBM alignof IBM
- sizeof
- IBM typeof
- C digitsof and precisionof C
- IBM __real__ and __imag__ IBM

Related reference:

"Pointer arithmetic" on page 101

"Lvalues and rvalues" on page 141

"Arithmetic conversions and promotions" on page 129

Increment operator ++

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. A pre-increment expression is an lvalue. A post-increment expression is an rvalue. For example:

```
play = ++play1 + play2++;
```

is similar to the following expressions; `play1` 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.

IBM C The increment operator has been extended to handle complex types. The operator works in the same manner as it does on a real type, except that only the real part of the operand is incremented, and the imaginary part is unchanged. C IBM

Decrement operator --

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. A pre-decrement expression is an lvalue. A post-decrement expression is an rvalue.

For example:

```
play = --play1 + play2--;
```

is similar to the following expressions; `play1` 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.

IBM C The decrement operator has been extended to handle complex types, for compatibility with GNU C. The operator works in the same manner as it does on a real type, except that only the real part of the operand is decremented, and the imaginary part is unchanged. C IBM

Unary plus operator +

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 operator -

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 operator !

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

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`. C++

The following two expressions are equivalent:

```
!right;
right == 0;
```

Related reference:

"Boolean types" on page 55

Bitwise negation operator ~

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
```

IBM C The bitwise negation operator has been extended to handle complex types. With a complex type, the operator computes the complex conjugate of the operand by reversing the sign of the imaginary part. C IBM

Related reference:

"Trigraph sequences" on page 38

Address operator &

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.

C It cannot have the storage class `register`. C

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;
```

C++ The ampersand symbol & is used in C++ to form declarators for lvalue references in addition to being the address operator. The meanings are related but not identical.

```
int target;
int &rTarg = target;  // rTarg is an lvalue reference to an integer.
                      // The reference is initialized to refer to target.
void f(int*& p);      // p is an lvalue 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. C++

Related reference:

"Indirection operator *" on page 155

"Pointers" on page 100

"References (C++ only)" on page 107

Indirection operator *

The * (indirection) operator determines the value referred to by the pointer-type operand. The operand can be a pointer to an incomplete type that is not cv void. The lvalue thus obtained cannot be converted to a C++11 prvalue C++11 rvalue. 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 C a function designator C C++ an lvalue referring to the object to which the operand points C++ . 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.

Related reference:

"Arrays" on page 104

"Pointers" on page 100

The typeid operator (C++ only)

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 runtime type identification (RTTI) support in C++.

typeid operator syntax

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.

Related reference:

"Type names" on page 99

"The typeof operator (IBM extension)" on page 159

The __alignof__ operator (IBM extension)

The `__alignof__` operator is a language extension to C99 and Standard C++ that returns the position to which its operand is aligned. The operand can be an expression or a parenthesized type identifier. If the operand is an expression that represents an lvalue, the number that is 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 that is usually required for the type on the target platform.

The `__alignof__` operator cannot be applied to the following situations:

- An lvalue that represents a bit field
- A function type
- An undefined structure or class
- An incomplete type (such as `void`)

__alignof__ operator syntax

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.

The 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.

sizeof operator syntax

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`.

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 can use the `sizeof` operator to determine the size that a data type represents. For example:

```
sizeof(int);
```

The `sizeof` operator applied to a type name yields the amount of memory that can be used by an object of that type, including any internal or trailing padding.

Using the `sizeof` operator with a fixed-point decimal type results in the total number of bytes that are occupied by the decimal type. z/OS XL C/C++ implements decimal data types using the native packed decimal format. Each digit occupies half a byte. The sign occupies an additional half byte. The following example gives you a result of 6 bytes:

```
sizeof(decimal(10,2));
```

For compound types, results are as follows:

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. It is equal to the number of bytes in an object of that class, also including any padding required for placing class objects in an array. C++
C++ A reference	The result is the size of the referenced object. C++

The sizeof operator cannot 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. 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. For example, 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, int, or any enumeration typeof the default enum size.

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.

C++11 sizeof... is a unary expression operator introduced by the variadic template feature. This operator accepts an expression that names a parameter pack as its operand. It then expands the parameter pack and returns the number of arguments provided for the parameter pack. Consider the following example:

```
template<typename...T> void foo(T...args){
    int v = sizeof...(args);
}
```

In this example, the variable *v* is assigned to the number of the arguments provided for the parameter pack args.

Notes:

- The operand of the sizeof... operator must be an expression that names a parameter pack.
- The operand of the sizeof operator cannot be an expression that names a parameter pack or a pack expansion.

For more information, see "Variadic templates (C++11)" on page 468

C++11

Related reference:

"Type names" on page 99

"Integer constant expressions" on page 144

"Arrays" on page 104

"References (C++ only)" on page 107

The typeof operator (IBM extension)

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. 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 alternate spelling of the keyword, __typeof__, is recommended.

typeof operator syntax

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

```
typedef 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; }
```

Notes:

1. The typeof and __typeof__ keywords are supported as follows:
   - The __typeof__ keyword is recognized in C under LANGLVL(EXTC89|EXTC99|EXTENDED), and in C++ under the LANGLVL(EXTENDED).
   - The typeof keyword is only recognized when the KEYWORD(TYPEOF) compiler option is in effect.

Related reference:

"The decltype(*expression*) type specifier (C++11)" on page 78

"Type names" on page 99

"typedef definitions" on page 74

The digitsof and precisionof operators (C only)

The digitsof and precisionof operators yield information about fixed-point decimal types or an expressions of the decimal type. The decimal.h header file defines the digitsof and precisionof macros.

The digitsof operator gives the number of significant digits of an object, and precisionof gives the number of decimal digits. That is,

```
digitsof(decimal(n,p)) = n
precisionof(decimal(n,p)) = p
```

The results of the digitsof and precisionof operators are integer constants.

Related reference:

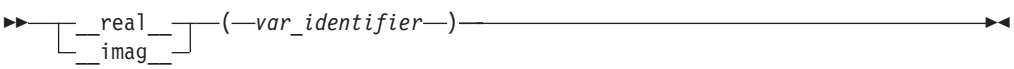
Fixed-point decimal literals (z/OS only)

"Fixed point decimal types (C only)" on page 58

The __real__ and __imag__ operators (IBM extension)

z/OS XL C/C++ extends the C99 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.

__real__ and __imag__ operator syntax



The *var_identifier* is the name of a previously declared complex variable. The __real__ operator returns the real part of the complex variable, while the __imag__ operator returns the imaginary part of the variable. If the operand of these operators is an lvalue, the resulting expression can be used in any context where lvalues are allowed. They are especially useful in initializations of complex variables, and as arguments to calls to library functions such as printf and scanf that have no format specifiers for complex types. For example:

```
float _Complex myvar;
__imag__(myvar) = 2.0f;
__real__(myvar) = 3.0f;
```

initializes the imaginary part of the complex variable myvar to 2.0*i* and the real part to 3.0, and

```
printf("myvar = %f + %f * i\n", __real__(myvar), __imag__(myvar));
```

prints:

```
myvar = 3.000000 + 2.000000 * i
```

Related reference:

Complex literals (C only)

Complex floating-point types (C only)

Binary expressions

A *binary expression* contains two operands separated by one operator.

- "Assignment operators" on page 161

All binary operators have left-to-right associativity, but not all binary operators have the same precedence. The ranking and precedence rules for binary operators is summarized in Table 30 on page 192.

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 descriptions of the operators, the usual arithmetic conversions are performed on the operands of most binary expressions.

The supported binary operators are as follows:

- "Assignment operators"
- "Multiplication operator *" on page 163
- "Division operator /" on page 163
- "Remainder operator %" on page 164
- "Addition operator +" on page 164
- "Subtraction operator -" on page 164
- "Bitwise left and right shift operators << >>" on page 165
- "Relational operators < > <= >=" on page 165
- "Equality and inequality operators == !=" on page 167
- "Bitwise AND operator &" on page 168
- "Bitwise exclusive OR operator ^" on page 168
- "Bitwise inclusive OR operator |" on page 169
- "Logical AND operator &&" on page 169
- "Logical OR operator ||" on page 170
- "Array subscripting operator [ ]" on page 171
-
- "Comma operator ," on page 172
- "Pointer to member operators .* ->* (C++ only)" on page 173

Related reference:

"Lvalues and rvalues" on page 141

"Arithmetic conversions and promotions" on page 129

Assignment operators

An *assignment expression* stores a value in the object designated by the left operand. There are two types of assignment operators:

- "Simple assignment operator =" on page 162
- "Compound assignment operators" on page 162

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.

C The result of an assignment expression is not an lvalue. C

C++ The result of an assignment expression is an lvalue. C++

All assignment operators have the same precedence and have right-to-left associativity.

Simple assignment operator =

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*.

If the left operand is not a class type, the right operand is implicitly converted to the type of the left operand. This converted type is 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 is called.

If the left operand is an object of reference type, the compiler assigns the value of the right operand to the object denoted by the reference.

IBM A packed structure or union can be assigned to a nonpacked structure or union of the same type. A nonpacked structure or union can be assigned to a packed structure or union of the same type.

If one operand is packed and the other is not, z/OS XL C/C++ remaps the layout of the right operand to match the layout of the left. This remapping of structures might degrade performance. For efficiency, when you perform assignment operations with structures or unions, you should ensure that both operands are either packed or nonpacked.

Note: If you assign pointers to structures or unions, the objects they point to must both be either packed or nonpacked. See "Initialization of pointers" on page 116 for more information on assignments with pointers. IBM

Compound assignment operators

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.

Related reference:

"Lvalues and rvalues" on page 141

"Pointers" on page 100

"Type qualifiers" on page 85

Multiplication operator *

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 operator /

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.

If either operand is negative, the result is rounded towards zero.

The usual arithmetic conversions on the operands are performed.

Remainder operator %

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.

Addition operator +

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;
```

Related reference:

"Pointer arithmetic" on page 101

"Pointer conversions" on page 137

Subtraction operator -

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.

Related reference:

"Pointer arithmetic" on page 101

"Pointer conversions" on page 137

Bitwise left and right shift operators << >>

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 operators < > <= >=

The relational operators compare two operands and determine the validity of a relationship. 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 type of the result is `int` and has the values `1` if the specified relationship is true, and `0` if false. C

C++ The type of the result is `bool` and has the values `true` or `false`. C++

The result is not an lvalue.

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 and inequality operators == !=

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. 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.

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`.

C++ The type of the result is `bool` and has the values `true` or `false`. 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 reference:

Simple assignment operator =

Bitwise AND operator &

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
```

Bitwise exclusive OR operator ^

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 reference:

"Trigraph sequences" on page 38

Bitwise inclusive OR operator |

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 reference:

"Trigraph sequences" on page 38

Logical AND operator &&

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 an 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`

Expression	Result
`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
```

Logical OR operator ||

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 an arithmetic or pointer type. The usual arithmetic conversions on each operand are performed. C

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`. C++

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`

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;
}
```

See the output of the above example:

```
a[0] = 10
a[1] = 20
a[2] = 30
```

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. C++

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;
         }
      }
   }
```

C C99 allows array subscripting on arrays that are not lvalues. The following example is valid in C99:

```
struct trio{int a[3];};
struct trio f();
foo (int index)
{
    return f().a[index];
}
```

C

Related reference:

"Pointers" on page 100

"Integral types" on page 54

"Lvalues and rvalues" on page 141

"Arrays" on page 104

"Overloading subscripting" on page 336

"Pointer arithmetic" on page 101

Comma operator ,

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.

C The result of a comma expression is not an lvalue. C

C++ 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
```

C++

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. Similarly, in the following example, the value of the expression:

```
intensity++, shade * increment, rotate(direction);
```

is the value of the expression:

```
rotate(direction)
```

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 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

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.
`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.

Pointer to member operators .* ->* (C++ only)

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 reference:

"Class member lists" on page 357

"Pointers to members" on page 362

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.

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))
```

Types in conditional C expressions (C only)

In C, a conditional expression is not an lvalue, nor is its result.

Table 26. Types of operands and results in conditional C expressions

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 composite 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

Types in conditional C++ expressions (C++ only)

Table 27. Types of operands and results in C++ conditional expressions

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`
Class `T`	Class `X`	Class type for which a conversion exists. If more than one possible conversion exist, 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 statement is equivalent to the previous expression.

```
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 lower 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);
```

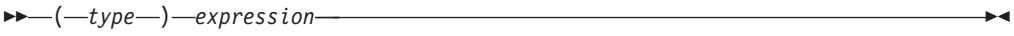
Cast expressions

A cast operator is used for *explicit type conversions*. It converts the value of an expression to a specified type.

The following cast operators are supported:

Cast operator ()

Cast expression syntax



C In C, the result of the cast operation is not an lvalue. C

C++ In C++, the cast result belongs to one of the following value categories:

- If `type` is an lvalue reference type C++11 or an rvalue reference to a function type C++11 , the cast result is an lvalue.
- C++11 If `type` is an rvalue reference to an object type, the cast result is an xvalue. C++11
- In all other cases, the cast result is a C++11 (prvalue) C++11 rvalue.

C++

The following example demonstrates the use of the cast operator to dynamically create 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 holds an object of the size of its argument. The statement `int* myArray = (int*) malloc(10 * sizeof(int))` has the following steps:

- 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`.

C++ In C++ you can also use the following objects in cast expressions:

- Function-style casts
- C++ conversion operators, such as `static_cast`.

Function-style notation converts the value of *expression* to the type *type*:

`type(expression)`

The following example shows the same value cast with a C-style cast, the C++ function-style cast, and a C++ cast operator:

```
#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;
}
```

See 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.

For C++, the operand of a cast expression 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. C++

Related reference:

"Structures and unions" on page 60

"Type names" on page 99

"Conversion functions" on page 426

"Conversion constructors" on page 424

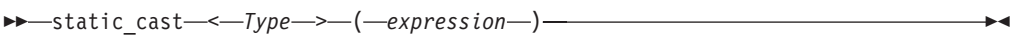
"Lvalues and rvalues" on page 141

"References (C++ only)" on page 107

The static_cast operator (C++ only)

The *static_cast* operator converts a given expression to a specified type.

static_cast operator syntax



C++11 With the right angle bracket feature, you may specify a `template_id` as `Type` in the `static_cast` operator with the `>>` token in place of two consecutive `>` tokens. For details, see "Class templates" on page 441. C++11

The result of `static_cast<Type>(expression)` belongs to one of the following value categories:

- If `Type` is an lvalue reference type C++11 or an rvalue reference to a function type C++11 , `static_cast<Type>(expression)` is an lvalue.
- C++11 If `Type` is an rvalue reference to an object type, `static_cast<Type>(expression)` is an xvalue. C++11
- In all other cases, `static_cast<Type>(expression)` is a C++11 (prvalue) C++11 rvalue.

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 output of this example is:

```
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 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 converts it to a null pointer value of the target type.

The compiler supports the following types of cast operations:

- An lvalue of type `A` to type `B&`, and the cast result is an lvalue of type `B`
- C++11 An lvalue or xvalue of type `A` to type `B&&`, and the cast result is an xvalue of type `B` C++11
- A C++11 (prvalue) C++11 rvalue of pointer to `A` to pointer to `B`
- C++11 An lvalue of type `A` to type `B&&` if an xvalue of type `A` can be bound directly to a reference of type `B&&` C++11
- An expression `e` to type `T` if the direct initialization `T t(e)` is valid.

To support the first three cast operations, the following conditions must be satisfied:

- A is a base class of B.
- There exists a standard conversion from a pointer to type B to a pointer to type A.
- Type B is the same as or more cv-qualified than type A.
- A is not a virtual base class or a base class of a virtual base class of B.

You can cast a C++11 (prvalue) C++11 rvalue of a pointer to member of A whose type is cv1 T to a C++11 (prvalue) C++11 rvalue of a pointer to member of B whose type is cv2 T if the following conditions are satisfied:

- B is a base class of A.
- There exists a standard conversion from a pointer to member of B whose type is T to a pointer to member of A whose type is T.
- cv2 is the same or more cv-qualification than cv1.

You can explicitly convert a pointer to cv1 void to a pointer to cv2 void if cv2 is the same or more cv-qualification than cv1.

Related reference:

"User-defined conversions" on page 423

"Type-based aliasing" on page 102

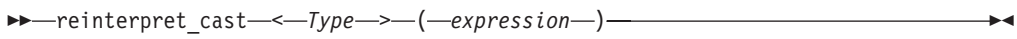
"Lvalues and rvalues" on page 141

"References (C++ only)" on page 107

The reinterpret_cast operator (C++ only)

A *reinterpret_cast* operator handles conversions between unrelated types.

reinterpret_cast operator syntax



C++11 With the right angle bracket feature, you may specify a template_id as Type in the reinterpret_cast operator with the >> token in place of two consecutive > tokens. For details, see "Class templates" on page 441. C++11

The result of reinterpret_cast<Type>(expression) belongs to one of the following value categories:

- If Type is an lvalue reference type C++11 or an rvalue reference to a function type C++11, reinterpret_cast<Type>(expression) is an lvalue.
- C++11 If Type is an rvalue reference to an object type, reinterpret_cast<Type>(expression) is an xvalue. C++11
- In all other cases, reinterpret_cast<Type>(expression) is a C++11 (prvalue) C++11 rvalue.

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 a type `T` and an lvalue expression `x`, the following two expressions for lvalue references have different syntax but the same semantics:

- `reinterpret_cast<T&>(x)`
- `*reinterpret_cast<T*>(&(x))`

C++11 Given a type `T` and an lvalue expression `x`, the following two expressions for rvalue references have different syntax but the same semantics:

- `reinterpret_cast<T&&>(x)`
- `static_cast<T&&>(*reinterpret_cast<T*>(&(x)))`

C++11

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 standard 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 executes as expected when compiled unoptimized in C++ or in K&R C or with NOANSIALIAS. It also successfully compiles optimized in C++ with ANSIALIAS, but does 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.

```
1  /* separately compiled file 1 */
2      extern float f;
3      extern int * int_pointer_to_f = (int *) &f; /* suspicious cast */
4
5  /* separately compiled file 2 */
6      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 reference:

"User-defined conversions" on page 423

"Lvalues and rvalues" on page 141

"References (C++ only)" on page 107

The const_cast operator (C++ only)

A *const_cast operator* adds or removes a `const` or `volatile` modifier to or from a type.

const_cast operator syntax

C++11 With the right angle bracket feature, you may specify a `template_id` as `Type` in the `const_cast` operator with the `>>` token in place of two consecutive `>` tokens. For details, see "Class templates" on page 441. C++11

The result of `const_cast<Type>(expression)` belongs to one of the following value categories:

- If `Type` is an lvalue reference to an object type, `const_cast<Type>(expression)` is an lvalue.
- C++11 If `Type` is an rvalue reference to an object type, `const_cast<Type>(expression)` is an xvalue. C++11
- In all other cases, `const_cast<Type>(expression)` is a C++11 (prvalue) C++11 rvalue.

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.

If a pointer to `T1` can be converted to a pointer to `T2` using `const_cast<T2>`, where `T1` and `T2` are object types, you can also make the following types of conversions:

- An lvalue of type `T1` to an lvalue of type `T2` using `const_cast<T2&>`
- C++11 An lvalue or xvalue of type `T1` to an xvalue of type `T2` using `const_cast<T2&&>` C++11
- C++11 A prvalue of class type `T1` to an xvalue of type `T2` using `const_cast<T2&&>` C++11

If a conversion from a C++11 (prvalue) C++11 rvalue of type pointer to `T1` to type pointer to `T2` casts away constness, the following types of conversions also cast away constness:

- An lvalue of type `T1` to an lvalue of type `T2`
- C++11 An expression of type `T1` to an xvalue of type `T2` C++11
- A C++11 (prvalue) C++11 rvalue of type pointer to data member of `X` of type `T1` to type pointer to data member of `Y` of type `T2`

Types cannot 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 does 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 does 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 does 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 reference:

"Type qualifiers" on page 85

"Type-based aliasing" on page 102

"Lvalues and rvalues" on page 141

"References (C++ only)" on page 107

The dynamic_cast operator (C++ only)

The *dynamic_cast* operator checks the following types of conversions at run time:

- A pointer to a base class to a pointer to a derived class
- An lvalue referring to a base class to an lvalue reference to a derived class

- C++11 An xvalue referring to a base class to an rvalue reference to a derived class C++11

A program can thereby use a class hierarchy safely. This operator and the `typeid` operator provide runtime type identification (RTTI) support in C++.

dynamic_cast operator syntax

C++11 With the right angle bracket feature, you may specify a `template_id` as `T` in the `dynamic_cast` operator with the `>>` token in place of two consecutive `>` tokens. For details, see "Class templates" on page 441. C++11

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`.

The following rules apply to the `dynamic_cast<T>(v)` expression:

- If `T` is a pointer type, `v` must be a C++11 (prvalue) C++11 rvalue, and `dynamic_cast<T>(v)` is a C++11 (prvalue) C++11 rvalue of type `T`.
- If `T` is an lvalue reference type, `v` must be an lvalue, and `dynamic_cast<T>(v)` is an lvalue of the type that is referred by `T`.
- C++11 If `T` is an rvalue reference type, `dynamic_cast<T>(v)` is an xvalue of the type that is referred by `T`. C++11

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>`.

If `T` is a void pointer, then `dynamic_cast` returns 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 is similar to the following result. Both `vp` and `&bobj` 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 a 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* might point to an object that is not a base class subobject of type A that belongs to an object of type B or a class derived from B. The dynamic_cast operator ensures that if you convert a pointer to class A to a pointer to class B, the object of type A pointed to by the former belongs to an object of type B or a class derived from B as a base class subobject.

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);
}
```

See 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 runtime type information generated from polymorphic classes.

Related reference:

"Derivation" on page 383

"User-defined conversions" on page 423

"Type-based aliasing" on page 102
"Lvalues and rvalues" on page 141
"References (C++ only)" on page 107

Compound literal expressions

A *compound literal* is a postfix expression that provides an unnamed object whose value is given by an initializer list. The C99 language feature allows you to pass parameters to functions without the need for temporary variables. It is useful for specifying constants of an aggregate type (arrays, structures, and unions) when only one instance of such types is needed. C++ To be compatible with C99, the z/OS XL C/C++ compiler supports this feature as an IBM extension. 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.

Compound literal syntax

The *type_name* can be any data type, including user-defined types. It can be an array of unknown size, but not a variable length array. If the type is an array of unknown size, the size is determined by the initializer list.

The following example passes a constant structure variable of type `point` containing two integer members to the function `drawline`:

```
drawline((struct point){6,7});
```

If the compound literal occurs outside the body of a function, the initializer list must consist of constant expressions, and the unnamed object has static storage duration. If the compound literal occurs within the body of a function, the initializer list need not consist of constant expressions, and the unnamed object has automatic storage duration.

IBM For compatibility with GNU C, a static variable can be initialized with a compound literal of the same type, provided that all the initializers in the initializer list are constant expressions. IBM

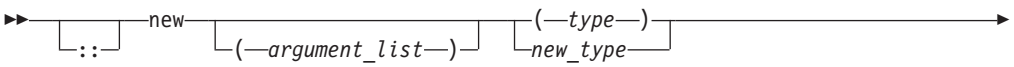
Related reference:

String literals

new expressions (C++ only)

The `new` operator provides dynamic storage allocation.

new operator syntax



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 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 integral constant expressions that evaluate to positive values. 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.

Related reference:

Generalized constant expressions (C++11)

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 reference:

"delete expressions (C++ only)" on page 189

"Scope resolution operator :: (C++ only)" on page 148

"Overview of constructors and destructors" on page 407

Initialization of objects created with the new operator

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 reference:

"Overview of constructors and destructors" on page 407

Handling new allocation failure

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.
```

delete expressions (C++ only)

The delete operator destroys the object created with new by deallocating the memory associated with the object.

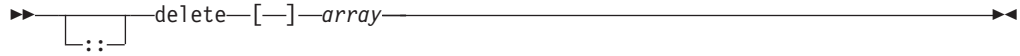
The delete operator has a void return type.

delete operator syntax

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.

delete[] operator syntax



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 reference:

"The void type" on page 59

"Overview of constructors and destructors" on page 407

throw expressions (C++ only)

A *throw* expression is used to throw exceptions to C++ exception handlers. A throw expression is of type `void`.

Related reference:

Chapter 16, "Exception handling (C++ only)," on page 485

"The void type" on page 59

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 tables list the C and C++ language operators in order of precedence and show the direction of associativity for each operator. Operators that have the same rank have the same precedence.

Table 28. Precedence and associativity of postfix 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*)

Table 29. 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*

Table 29. Precedence and associativity of unary operators (continued)

Rank	Right associative?	Operator function	Usage
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*
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*

Table 30. Precedence and associativity of binary operators

Rank	Right associative?	Operator function	Usage
4		C++ member selection	*object .* ptr_to_member*
4		C++ 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*

Table 30. Precedence and associativity of binary operators (continued)

Rank	Right associative?	Operator function	Usage
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*

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.

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();
```

The order of evaluation for function call arguments or for the operands of binary operators is not specified. Therefore, the following expressions are ambiguous:

```
z = (x * ++y) / func1(y);
func2(++i, x[i]);
```

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()`.

Reference collapsing (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

Before C++11, references to references are ill-formed in the C++ language. In C++11, the rules of reference collapsing apply when you use references to references through one of the following contexts:

- A `decltype` specifier
- A `typedef` name
- A template type parameter

You can define a variable `var` whose declared type `TR` is a reference to the type `T`, where `T` is also a reference type. For example,

```
// T denotes the int& type
typedef int& T;

// TR is an lvalue reference to T
```

```
typedef T& TR;

// The declared type of var is TR
TR var;
```

The actual type of var is listed in the following table for different cases, where neither TR nor T is qualified by cv-qualifiers.

Table 31. Reference collapsing without cv-qualifiers

T	TR	Type of var
A	T	A[1]
A	T&	A&[1]
A	T&&	A&&[1]
A&	T	A&[1]
A&	T&	A&
A&	T&&	A&
A&&	T	A&&[1]
A&&	T&	A&
A&&	T&&	A&&

Note:

1. Reference collapsing does not apply in this case, because T and TR are not both reference types.

The general rule in this table is that when T and TR are both reference types, but are not both rvalue reference types, var is of an lvalue reference type.

Example 1

```
typedef int& T;

// a has the type int&
T&& a;
```

In this example, T is of the int& type, and the declared type of a is T&&. After reference collapsing, the type of a is int&.

Example 2

```
template <typename T> void func(T&& a);
auto fp = func<int&&>;
```

In this example, the actual parameter of T is of the int&& type, and the declared type of a is T&&. An rvalue reference to an rvalue reference is formed. After reference collapsing, the type of a is int&&.

Example 3

```
auto func(int& a) -> const decltype(a)&;
```

In this example, decltype(a), which is a trailing return type, refers to the parameter a, whose type is int&. After reference collapsing, the return type of func is int&.

You can define a variable var whose declared type TR is a reference to the type T, where T is also a reference type. If either TR or T is qualified by cv-qualifiers, then the actual type of var is listed in the following table for different cases.

Table 32. Reference collapsing with cv-qualifiers

T	TR	Type of var
A	const T	const A[1]
const A	volatile T&	const volatile A&[1]
A	const T&&	const A&&[1]
A&	const T	A&[1]
const A&	volatile T&	const A&
const A&	T&&	const A&
A&&	const T	A&&[1]
const A&&	volatile T&	const A&
const A&&	T&&	const A&&

Note:

1. Reference collapsing does not apply in this case, because T and TR are not both reference types.

The general rule of this table is that when T is a reference type, the type of var inherits only the cv-qualifiers from T.

Related reference:

"The decltype(*expression*) type specifier (C++11)" on page 78

"typedef definitions" on page 74

"Type template parameters" on page 434

Chapter 7. Statements

A statement, the smallest independent computational unit, specifies an action to be performed. In most cases, statements are executed in sequence.

The following list is a summary of the statements available in C and C++:

- "Labeled statements"
- "Expression statements" on page 198
- "Block statements" on page 199
- "Selection statements" on page 200
- "Iteration statements" on page 206
- "Jump statements" on page 210
- "Null statement" on page 215
- "Inline assembly statements (IBM extension)" on page 215

Related reference:

Chapter 3, "Data objects and declarations," on page 41

"Function declarations" on page 219

"try blocks" on page 485

Labeled statements

There are three kinds of labels: identifier, case, and default.

Labeled statement syntax

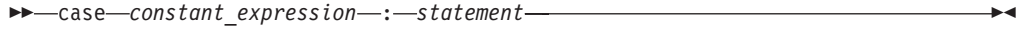
The label consists of the *identifier* and the colon (**:**) character.

C A label name must be unique within the function in which it appears. C

C++ In C++, an identifier label can 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 namespace; you do not have to worry about identifier labels conflicting with other identifiers. However, you cannot redeclare a label within a function. C++

Case and default label statements only appear in `switch` statements. These labels are accessible only within the closest enclosing `switch` statement.

case statement syntax



default statement syntax

The following are examples of labels:

```
comment_complete : ;            /* null statement label */
test_for_null : if (NULL == pointer)
```

Related reference:

"The goto statement" on page 213

"The switch statement" on page 202

Labels as values (IBM extension)

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 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;
   }
   ...
}
```

Expression statements

An *expression statement* contains an expression. The expression can be null.

Expression statement syntax

An expression statement evaluates *expression*, then discards the value of the expression. An expression statement without an expression is a null statement.

See the following examples of statements:

```
printf("Account Number: \n");          /* call to the printf    */
marks = dollars * exch_rate;                /* assignment to marks     */
(difference < 0) ? ++losses : ++gain;  /* conditional increment */
```

Related reference:

Chapter 6, "Expressions and operators," on page 141

Resolution of ambiguous statements (C++ only)

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 reference:

Chapter 3, “Data objects and declarations,” on page 41

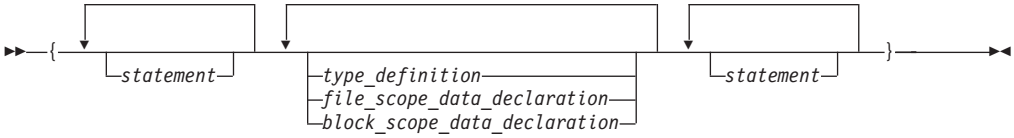
Chapter 6, “Expressions and operators,” on page 141

“Function call expressions” on page 149

Block statements

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.

Block statement syntax



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.

Related reference:

"Command-line arguments" on page 248

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`.

Selection statements

Selection statements consist of the following types of statements:

- The if statement
- The switch statement

The if statement

An `if` statement is a selection statement that allows more than one possible flow of control.

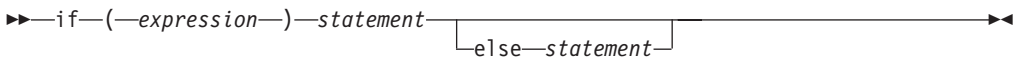
C++ An `if` statement lets you conditionally process a statement when the specified test expression, implicitly converted to `bool`, evaluates to `true`. If the implicit conversion to `bool` fails the program is ill-formed. C++

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. C

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.

if statement syntax



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)
      fxph = furlongs/kegs;
}
else
   fxph = 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;
```

Related reference:

"Boolean types" on page 55

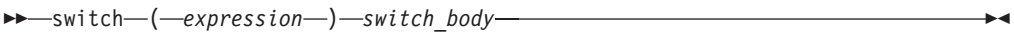
The switch statement

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

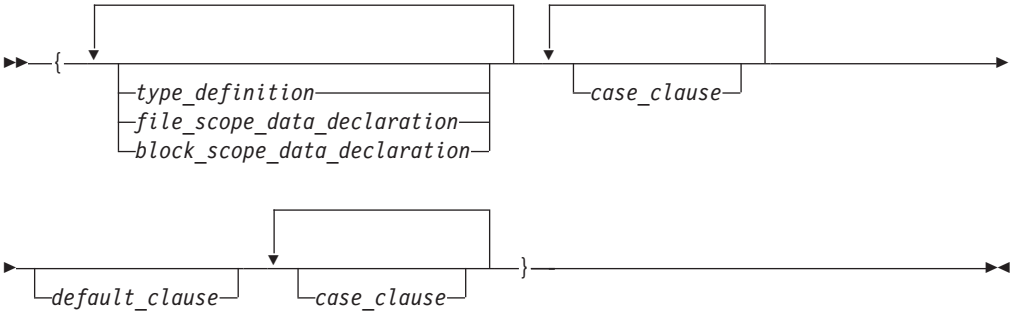
- A `case` label
- An optional `default` label
- A `case` expression
- A list of statements.

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.

switch statement syntax



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.



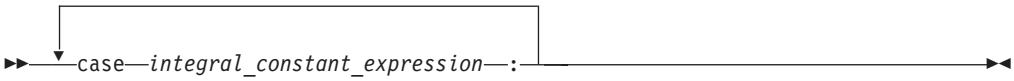
Note: An initializer within a *type_definition*, *file_scope_data_declaration* or *block_scope_data_declaration* is ignored.

A *case clause* contains a *case label* followed by any number of statements. A case clause has the form:

Case clause syntax

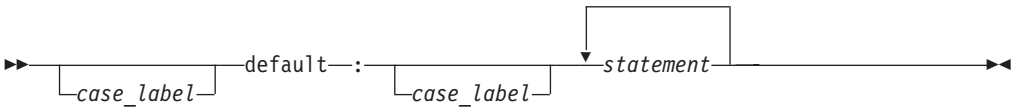
A *case label* contains the word `case` followed by an integral constant expression and a colon. The value of each integral constant expression must represent a different value; you cannot have duplicate `case` labels. Anywhere you can put one `case` label, you can put multiple `case` labels. A case label has the form:

case label syntax



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:

Default clause statement



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 on switch statements

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

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. C++

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:

CCNRAB1

```
/**
 ** 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.

Related reference:

Case and default labels

"The break statement" on page 210

"Generalized constant expressions (C++11)" on page 149

Iteration statements

Iteration statements consist of the following types of statements:

- The while statement
- The do statement
- The for statement

Related reference:

"Boolean types" on page 55

The while statement

A *while statement* repeatedly runs the body of a loop until the controlling expression evaluates to `false` (or `0` in C).

while statement syntax

C The *expression* must be of arithmetic or pointer type. C

C++ The expression must be convertible to `bool`. C++

The expression is evaluated to determine whether or not to process the body of the loop. 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. C++

In the following example, `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.

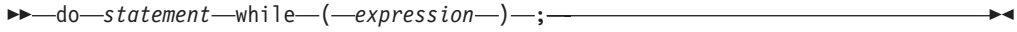
CCNRAA7

```
/**
 ** 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);
}
```

The 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.

do statement syntax



C The *expression* must be of arithmetic or pointer type. C

C++ The controlling *expression* must be convertible to type `bool`. C++

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. C++

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;
}
```

See the following output of the above example:

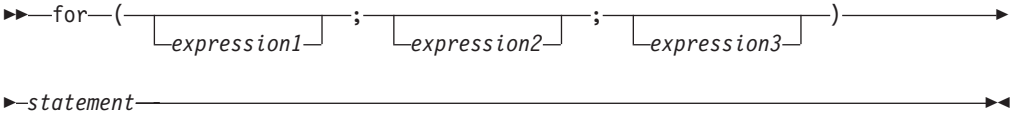
```
Value of i: 1
Value of i: 2
Value of i: 3
Value of i: 4
Value of i: 5
```

The for statement

A *for statement* provides the following benefits:

- 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).

for statement syntax



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. You can also use this expression to declare a variable, provided that the variable is not declared as static (it must be automatic and may also be declared as `register`). 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. 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 *expression2* must be of arithmetic or pointer type. C C++ *expression2* must be convertible to type `bool`. C++

If *expression2* evaluates to C `0` C or C++ `false` 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.

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.

Jump statements

Jump statements consist of the following types of statements:

- "The break statement"
- "The continue statement"
- "The return statement" on page 212
- "The goto statement" on page 213

The 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.

break statement syntax

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.

The 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:

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.

CCNRAA3

```
/**
 ** 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.

CCNRAA4

```
/**
 ** 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);
}
```

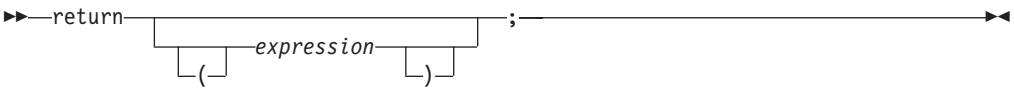
The program produces the following output:

```
letter count = 5
```

The return statement

A *return statement* ends the processing of the current function and returns control to the caller of the function.

return statement syntax



A value-returning function should include a `return` statement, containing an *expression*.

C If an expression is not given on a `return` statement in a function declared with a non-`void` return type, the compiler issues a warning message. C

C++ If an expression is not given on a `return` statement in a function declared with a non-`void` return type, the compiler issues an error message. C++

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.

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.

C++ If a `return` statement is used, it must not contain an expression. C++

Examples of return statements

The following are 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);
}
```

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.

Related reference:

"Function return type specifiers" on page 234

"Function return values" on page 235

The goto statement

A `goto` statement causes your program to unconditionally transfer control to the statement that is associated with the label specified on the `goto` statement.

goto statement syntax

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. C

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.

CCNRAA6

```
/**
 ** 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 statement (IBM extension)

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 extension to C99 and C++, implemented to facilitate porting programs developed with GNU C.

Computed goto statement syntax

The **expression* is an expression of type void*.

Related reference:

"Labeled statements" on page 197

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.

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 */
}
```

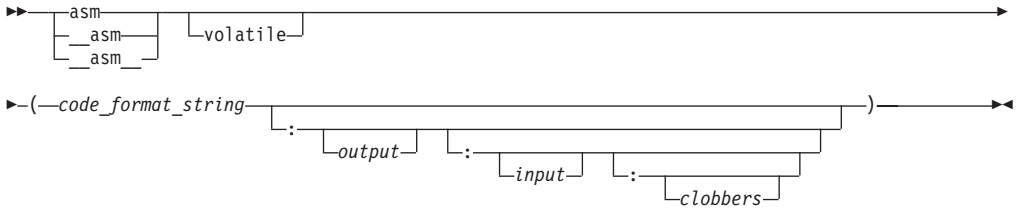
Inline assembly statements (IBM extension)

When the ASM compiler option is in effect, the compiler provides support for embedded assembly code fragments among C and C++ source statements. This extension allows programs to invoke IBM MVS™ system services directly via system-provided assembly macros.

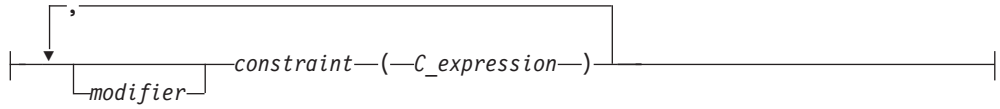
Note: For C source files, when the METAL or GENASM option is in effect, the compiler provides support for embedded assembly code fragments among C source statements.

The syntax is as follows:

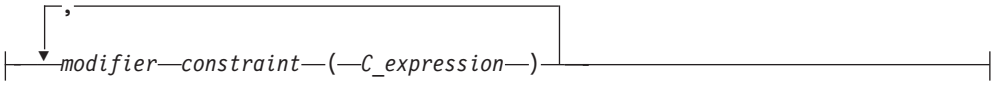
asm statement syntax — statement in local scope



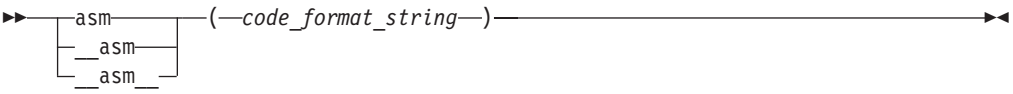
input:



output:



asm statement syntax — statement in global scope



volatile
The qualifier `volatile` instructs the compiler to perform only minimal optimizations on the assembly block. The compiler cannot move any instructions across the implicit fences surrounding the assembly block.

code_format_string
The *code_format_string* is the source text of the `asm` instructions and is a string literal similar to a `printf` format specifier.

output
The *output* consists of zero, one or more output operands, separated by commas. Each operand consists of a *constraint*(*C_expression*) pair. The output operand must be constrained by the `=` or `+` modifier (described below).

input The *input* consists of zero, one or more input operands, separated by commas. Each operand consists of a *constraint*(*C_expression*) pair.

clobbers

clobbers is a comma-separated list of register names enclosed in double quotes. If an `asm` instruction updates registers that are not listed in the *input* or *output* of the `asm` statement, the registers must be listed as clobbered registers. The following register names are valid :

r0 or R0 to r15 or R15
General purpose registers

modifier

The *modifier* can be one of the following operators:

= Indicates that the operand is write-only for this instruction. The previous value is discarded and replaced by output data.

+ Indicates that the operand is both read and written by the instruction.

& Indicates that the 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.

Note: The & modifier is ignored in z/OS V1R9.

constraint

The *constraint* is a string literal that describes the kind of operand that is permitted, one character per constraint. The following constraints are supported:

a Use an address register (general purpose register except `r0`).

d Use a data register that is an arbitrary general purpose register. This constraint is the same as the **r** constraint.

g Use a general register, memory, or immediate operand.

i Use an immediate integer or string literal operand.

m Use a memory operand supported by the machine.

n Use an immediate integer.

o Use a memory operand that is offsetable.

r Use a general register.

s Use a string literal operand.

0, 1, ...8, 9
A matching constraint. Allocate the same register in output as in the corresponding input.

I, J, K
Constant values. Fold the expression in the operand and substitute the value into the `%` specifier.

- I — signed 16-bit
- J — unsigned 16-bit shifted left 16 bits
- K — unsigned 16-bit constant

XL Only valid for METAL. Use only the parameter constraints listed in this constraint. `XL` is an optional prefix, followed by a colon (:), to introduce any of the following parameter constraints:

DS Do not generate a definition for the operand defined in the assembly statement; instead, substitute an assembly instruction to define the operand. Optionally, to specify the data size of the operand defined in the assembly statement, use a colon (:) followed by a positive integer. If you do not specify a data size, the size specified in the ASMDATASIZE option is used.

RP The operand requires a register pair. Optionally, to specify the constraint for the register pair, specify a **:**, followed by the *register_type*, optionally followed by another **:** and an optional *register_pair_flag*. The *register_pair_flag* can be one of the following:

o The operand needs an odd/even register pair.

e The operand needs an even/odd register pair.

If you do not specify a register type, `r` (general purpose register) is used as the default. If you do not specify a register pair flag, `e` (even/odd pair) is used as the default.

NR Use the named general purpose register. Use a colon (**:**) followed by the general purpose register name (see below for acceptable register names).

Note: The `XL` constraints can be used for both input and output operands, with the exception of DS, which can only be used for output operands.

C_expression

The *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 *C_expression* must be consistent with the constraint specified on it. For example, if `i` is specified, the operand must be an integer constant number.

Note: If pointer expressions are used in *input* or *output*, the assembly instructions honor the ANSI aliasing rule (see "Type-based aliasing" on page 102 for more information). This means that indirect addressing using values in pointer expression operands should be consistent with the pointer types; otherwise, you must disable the ANSIALIAS option during compilation.

For more information about ASM and ANSIALIAS options, see ASM and ANSIALIAS options in the *z/OS XL C/C++ User's Guide*.

Restrictions on inline assembly statements

The following restrictions are on the use of inline assembly statements:

- 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.
- If an `asm` statement is used to define data, it cannot contain assembly instructions for other purposes.
- Only `asm` statements that are used to define data can exist in global scope.
- The XL:* constraints are only supported for metal C programs.
- Each assembly statement can define only one variable.
- The symbol used in the assembly statement must be unique within the scope of the source file and is valid according to the assembler's requirements.
- Using registers that are reserved (for example, killing a register used by the linkage) is not supported.
- Referencing an external symbol directly without going through the operand list is not supported.

Related reference:

Variables in specified registers (IBM extension)

Chapter 8. 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 might not produce consistent outputs for all inputs, might not produce output at all, or might have side effects. Functions can be understood as user-defined operations, in which the parameters of the parameter list, if any, are the operands.

Function declarations and definitions

The distinction between a function *declaration* and function *definition* is similar to that of a data declaration and definition. The declaration establishes the names and characteristics of a function but does not allocate storage for it, while the *definition* specifies the body for a function, associates an identifier with the function, and allocates storage for it. Thus, the identifiers declared in this example:

```
float square(float x);
```

do not allocate storage.

The *function definition* contains a function declaration and the body of a function. The body is a block of statements that perform the work of the function. The identifiers declared in this example allocate storage; they are both declarations and definitions.

```
float square(float x)
{ return x*x; }
```

A function can be declared several times in a program, but all declarations for a given function must be compatible; that is, the return type is the same and the parameters have the same type. However, a function can only have one definition. Declarations are typically placed in header files, while definitions appear in source files.

Function declarations

A function identifier preceded by its return type and followed by its parameter list is called a *function declaration* or *function prototype*. The prototype informs the compiler of the format and existence of a function prior to its use. 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.

C++ Implicit declaration of functions is not allowed: you must explicitly declare every function before you can call it. C++

C If a function declaration is not visible at the point at which a call to the function is made, the compiler assumes an implicit declaration of `extern int func();` However, for conformance to C99, you should explicitly prototype every function before making a call to it. C

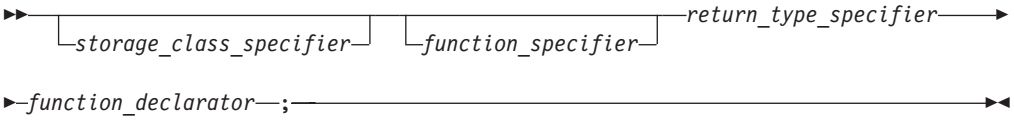
The elements of a declaration for a function are as follows:

- "Function storage class specifiers" on page 225, which specify linkage

- "Function return type specifiers" on page 234, which specify the data type of a value to be returned
- "Function specifiers" on page 227, which specify additional properties for functions
- "Function declarators" on page 235, which include function identifiers as well as lists of parameters

All function declarations have the form:

Function declaration syntax



C++11

Note: When *function_declarator* incorporates a trailing return type, *return_type_specifer* must be `auto`. For more information about trailing return type, see "Trailing return type (C++11)" on page 239.

C++11

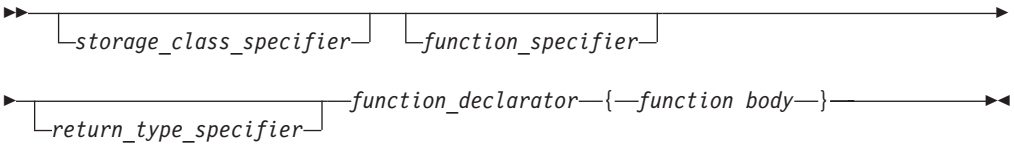
Function definitions

The elements of a function definition are as follows:

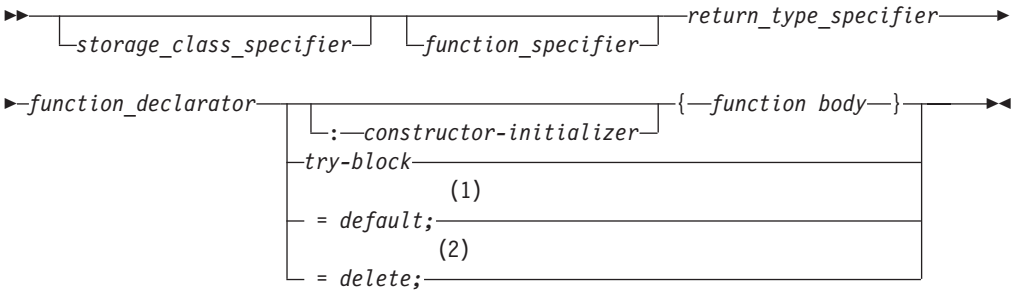
- "Function storage class specifiers" on page 225, which specify linkage
- "Function return type specifiers" on page 234, which specify the data type of a value to be returned
- "Function specifiers" on page 227, which specify additional properties for functions
- "Function declarators" on page 235, which include function identifiers as well as lists of parameters
- The *function body*, which is a braces-enclosed series of statements representing the actions that the function performs
- C++ Constructor-initializers, which are used only in constructor functions declared in classes; they are described in "Constructors" on page 408.
- Try blocks, which are used in class functions; they are described in "try blocks" on page 485. C++

Function definitions take the following form:

Function definition syntax (C only)



Function definition syntax (C++ only)



Notes:

1 This syntax is valid only in the C++11 standard.

2 This syntax is valid only in the C++11 standard.

C++11

Note: When *function_declarator* incorporates a trailing return type, *return_type_specifer* must be `auto`. For more information about trailing return type, see "Trailing return type (C++11)" on page 239.

C++11

Explicitly defaulted functions

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

Explicitly defaulted function declaration is a new form of function declaration that is introduced into the C++11 standard. You can append the `=default;` specifier to the end of a function declaration to declare that function as an explicitly defaulted function. The compiler generates the default implementations for explicitly defaulted functions, which are more efficient than manually programmed function implementations. A function that is explicitly defaulted must be a special member function and has no default arguments. Explicitly defaulted functions can save your effort of defining those functions manually.

You can declare both inline and out-of-line explicitly defaulted functions. For example:

```
class A{
public:
   A() = default;         // Inline explicitly defaulted constructor definition
   A(const A&);
   ~A() = default;        // Inline explicitly defaulted destructor definition
};

A::A(const A&) = default; // Out-of-line explicitly defaulted constructor definition
```

You can declare a function as an explicitly defaulted function only if the function is a special member function and has no default arguments. For example:

```
class B {
public:
  int func() = default;  // Error, func is not a special member function.
  B(int, int) = default; // Error, constructor B(int, int) is not
                         // a special member function.
  B(int=0) = default;    // Error, constructor B(int=0) has a default argument.
};
```

The explicitly defaulted function declarations enable more opportunities in optimization, because the compiler might treat explicitly defaulted functions as trivial.

Related reference:

Deleted functions (C++11)

Chapter 14, “Special member functions (C++ only),” on page 407

Deleted functions

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

Deleted function declaration is a new form of function declaration that is introduced into the C++11 standard. To declare a function as a deleted function, you can append the `=delete;` specifier to the end of that function declaration. The compiler disables the usage of a deleted function.

You can declare an implicitly defined function as a deleted function if you want to prevent its usage. For example, you can declare the implicitly defined copy assignment operator and copy constructor of a class as deleted functions to prevent object copy of that class.

```
class A{
public:
  A(int x) : m(x) {}
  A& operator = (const A &) = delete;  // Declare the copy assignment operator
                                       // as a deleted function.
  A(const A&) = delete;                // Declare the copy constructor
                                       // as a deleted function.

private:
  int m;
};

int main(){
  A a1(1), a2(2), a3(3);
  a1 = a2;     // Error, the usage of the copy assignment operator is disabled.
  a3 = A(a2);  // Error, the usage of the copy constructor is disabled.
}
```

You can also prevent problematic conversions by declaring the undesirable conversion constructors and operators as deleted functions. The following example shows how to prevent undesirable conversions from `double` to a class type.

```
class B{
public:
  B(int){}
```

```
  B(double) = delete;  // Declare the conversioin constructor as a deleted function
};

int main(){
  B b1(1);
  B b2(100.1);         // Error, conversion from double to class B is disabled.
}
```

A deleted function is implicitly inline. A deleted definition of a function must be the first declaration of the function. For example:

```
class C {
public:
  C();
};

C::C() = delete;  // Error, the deleted definition of function C must be
                  // the first declaration of the function.
```

Related reference:

Explicitly defaulted functions (C++11)

Examples of function declarations

The following code fragments show several function declarations (or *prototypes*). The first declares a function `f` that takes two integer arguments and has a return type of `void`:

```
void f(int, int);
```

This 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);
```

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`.

```
extern int  f2(const int, ...);
```

C++ 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);
```

C++

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.

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;
}
```

Compatible functions (C only)

For two function types to be compatible, they must meet the following requirements:

- They must agree in the number of parameters (and use of ellipsis).
- They must have compatible return types.
- The corresponding parameters must be compatible with the type that results from the application of the default argument promotions.

The composite type of two compatible function types is determined as follows:

- 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 function types have parameter type lists, the composite type of each parameter is determined as follows:
  - The composite of parameters of different rank is the type that results from the application of the default argument promotions.
  - The composite of parameters with array or function type is the adjusted type.
  - The composite of parameters with qualified type is the unqualified version of the declared type.

For example, for the following two function declarations:

```
int f(int (*)(), double (*)[3]);
int f(int (*)(char *), double (*)[]);
```

The resulting composite type would be:

```
int f(int (*)(char *), double (*)[3]);
```

If the function declarator is not part of the function declaration, 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:

```
int myMin(int n, int m, int a[n][m]);
int myMin(int n, int m, int a[*][*]);
int myMin(int n, int m, int a[ ][*]);
int myMin(int n, int m, int a[ ][m]);
```

Related reference:

Compatible and composite types

Multiple function declarations (C++ only)

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 qualifier 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 reference:

"Overloading functions" on page 327

Function storage class specifiers

For a function, the storage class specifier determines the linkage of the function. By default, function definitions have external linkage, and can be called by functions defined in other files. C An exception is inline functions, which are treated by default as having internal linkage; see "Linkage of inline functions" on page 229 for more information. C

A storage class specifier may be used in both function declarations and definitions. The only storage class options for functions are:

- `static`
- `extern`

The static storage class specifier

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 `static` storage class specifier can be used in a function declaration only if it is at file scope. You cannot declare functions within a block as `static`.

C++ This use of `static` is deprecated in C++. Instead, place the function in the unnamed namespace. C++

Related reference:

"Internal linkage" on page 7

Chapter 9, "Namespaces (C++ only)," on page 317

The extern storage class specifier

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. The keyword `extern` is optional; if you do not specify a storage class specifier, the function is assumed to have external linkage.

C++ An `extern` declaration cannot appear in class scope. You can use the `extern` keyword with arguments that specify the type of linkage.

extern function storage class specifier syntax

where *linkage_specification* can be any of the following:

- builtin
- C
- C++
- COBOL
- FORTRAN
- OS
- OS_DOWNSTACK
- OS_NOSTACK
- OS_UPSTACK
- PLI

For an explanation of these options, see the descriptions in "#pragma linkage (C only)" on page 566.

The following fragments illustrate the use of `extern "C"` :

```
extern "C" int cf();      //declare function cf to have C linkage

extern "C" int (*c_fp)(); //declare a pointer to a function,
                          // called c_fp, which has C linkage

extern "C" {
  typedef void(*cfp_T)(); //create a type pointer to function with C
                          // linkage
```

```
  void cfn();             //create a function with C linkage
  void (*cfp)();          //create a pointer to a function, with C
                          // linkage
}
```

Linkage compatibility affects all C library functions that accept a user function pointer as a parameter, such as `qsort`. Use the `extern "C"` linkage specification to ensure that the declared linkages are the same. The following example fragment uses `extern "C"` with `qsort`.

```
#include <stdlib.h>

// function to compare table elements
extern "C" int TableCmp(const void *, const void *); // C linkage
extern void * GenTable();                            // C++ linkage

int main() {
  void *table;

  table = GenTable();               // generate table
  qsort(table, 100, 15, TableCmp);  // sort table, using TableCmp
                                    // and C library routine qsort();
}
```

While the C++ language supports overloading, other languages do not. The implications of this are:

- You can overload a function as long as it has C++ (default) linkage. Therefore, z/OS XL C++ allows the following series of statements:

  ```
  int func(int);            // function with C++ linkage
  int func(char);           // overloaded function with C++ linkage
  ```

  By contrast, you cannot overload a function that has non-C++ linkage:

  ```
  extern "FORTRAN"{int func(int);}
  extern "FORTRAN"{int func(int,int);}  // not allowed
                                        //compiler will issue an error message
  ```

- Only one non-C++-linkage function can have the same name as overloaded functions. For example:

  ```
  int func(char);
  int func(int);
  extern "FORTRAN"{int func(int,int);}
  ```

  However, the non-C++-linkage function cannot have the same parameters as any of the C++ functions with the same name:

  ```
  int func(char);      // first function with C++ linkage
  int func(int, int);  // second function with C++ linkage
  extern "FORTRAN"{int func(int,int);}  // not allowed since the parameter
                                        // list is the same as the one for
                                        // the second function with C++ linkage
                                        // compiler will issue an error message
  ```

C++

Related reference:

"External linkage" on page 8

"Class scope (C++ only)" on page 4

Chapter 9, "Namespaces (C++ only)," on page 317

Function specifiers

The available function specifiers for function definitions are:

- C++11 `constexpr`, which can be used to declare constexpr functions and constexpr constructors, and is described in "The constexpr specifier (C++11)" on page 83. C++11
- `inline`, which instructs the compiler to expand a function definition at the point of a function call.
- C++ `__cdecl`, which sets linkage conventions for C++ function calls to C functions. C++
- C++ `_Export`, which makes function definitions available to other modules. C++
- C++ `explicit`, which can only be used for member functions of classes, and is described in "Explicit conversion constructors" on page 425. C++
- C11 `_Noreturn`, which indicates that a function does not return to its caller. C11
- C++ `virtual`, which can only be used for member functions of classes, and is described in "Virtual functions" on page 399. C++

The inline function specifier

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. Using 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.

C Any function, with the exception of `main`, can be declared or defined as inline with the `inline` function specifier. Static local variables are not allowed to be defined within the body of an inline function. C

C++ C++ functions implemented inside of a class declaration are automatically defined inline. Regular C++ functions and member functions declared outside of a class declaration, with the exception of `main`, can be declared or defined as inline with the `inline` function specifier. Static locals and string literals defined within the body of an inline function are treated as the same object across translation units; see "Linkage of inline functions" on page 229 for details. C++

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.

The most efficient way to code an inline function is to place the inline function definition in a header file, and then include the header in any file containing a call to the function which you would like to inline.

Note: C To enable the `inline` function specifier in C, you must compile with **c99** or the LANGLVL(STDC99) or LANGLVL(EXTC99) options. C

Linkage of inline functions

C

In C, inline functions are treated by default as having *static* linkage; that is, they are only visible within a single translation unit. Therefore, in the following example, even though function `foo` is defined in exactly the same way, `foo` in file `a.c` and `foo` in file `b.c` are treated as separate functions: two function bodies are generated, and assigned two different addresses in memory:

```
// a.c

#include <stdio.h>

inline int foo(){
   return 3;
}

void g() {
   printf("foo called from g: return value = %d, address = %p\n", foo(), &foo);
}


// b.c

#include <stdio.h>

inline int foo(){
   return 3;
}

void g();

int main() {
   printf("foo called from main: return value = %d, address = %p\n", foo(), &foo);
   g();
}
```

The output from the compiled program is:

```
foo called from main: return value = 3, address = 0x10000580
foo called from g: return value = 3, address = 0x10000500
```

Since inline functions are treated as having internal linkage, an inline function definition can co-exist with a regular, external definition of a function with the same name in another translation unit. However, when you call the function from the file containing the inline definition, the compiler may choose *either* the inline version defined in the same file *or* the external version defined in another file for the call; your program should not rely on the inline version being called. In the following example, the call to `foo` from function `g` could return either 6 or 3:

```
// a.c

#include <stdio.h>

inline int foo(){
   return 6;
}

void g() {
   printf("foo called from g: return value = %d\n", foo());
}


// b.c
```

```
#include <stdio.h>

int foo(){
   return 3;
}

void g();

int main() {
   printf("foo called from main: return value = %d\n", foo());
   g();
}
```

Similarly, if you define a function as `extern inline`, or redeclare an `inline` function as `extern`, the function simply becomes a regular, external function and is not inlined.

C

Related reference:

"The static storage class specifier" on page 226

"The extern storage class specifier" on page 226

The _Noreturn function specifier

The `_Noreturn` function specifier declares a function that does not return to its caller. When you declare a function with the specifier, the compiler can better optimize your code without regard to what happens if it returns. Any function, with the exception of `main`, can be declared or defined with the `_Noreturn` function specifier.

When `_Noreturn` is enabled, the `__IBMC_NORETURN` macro is defined as 1.

Include the standard header file `stdnoreturn.h` in your program when using the `_Noreturn` function specifier.

The following code fragment shows a function definition with the `_Noreturn` specifier:

```
_Noreturn void f (void) {
  abort();
}
```

Notes:

- C The `_Noreturn` keyword is recognized under compilation with the **-qlanglvl=extc89**, **-qlanglvl=extc99**, **-qlanglvl=extended**, or **-qlanglvl=extc1x** compiler option. C
- C++ The `_Noreturn` keyword is recognized under compilation with the **-qlanglvl=extended**, **-qlanglvl=extended0x**, or **-qlanglvl=c1xnoreturn** compiler option. C++
- You can also define your own functions that never return by using the `_Noreturn` function specifier. However, any functions that are declared with `_Noreturn` must call one of the following functions. Otherwise, the functions will return the control to their respective caller.
  - `abort`
  - `exit`

- _Exit
- longjmp
- quick_exit
- thrd_exit

The __cdecl function specifier (C++ only)

You can use the __cdecl keyword to set linkage conventions for function calls in C++ applications. The __cdecl keyword instructs the compiler to read and write a parameter list by using C linkage conventions.

To set the __cdecl calling convention for a function, place the linkage keyword immediately before the function name or at the beginning of the declarator. For example:

```
void __cdecl f();
char (__cdecl *fp) (void);
```

z/OS XL C++ allows the __cdecl keyword on member functions and nonmember functions. These functions can be static or nonstatic. It also allows the keyword on pointer-to-member function types and the typedef specifier.

Note: The compiler accepts both _cdecl and __cdecl (both single and double underscore).

Following is an example:

```
// C++ nonmember functions
void __cdecl f1();
static void __cdecl f2();

// pointer to member function type
char (__cdecl *A::mfp) (void);

// typedef
typedef  void (* _cdecl void_fcn)(int);
// C++ member functions
class A {
  public:
    void __cdecl func();
    static void __cdecl func1();
}

// Template member functions
template <class T> X {
  public:
    void __cdecl func();
    static void __cdecl func1();
}

// Template functions
template <class T> T __cdecl foo(T i) {return i+1;}
template <class T> T static _cdecl foo2(T i) {return i+1;}
```

The __cdecl linkage keyword only affects parameter passing; it does not prevent function name mangling. Therefore, you can still overload functions with non-default linkage. Note that you only acquire linkage by explicitly using the __cdecl keyword. It overrides the linkage that it inherits from an extern "linkage" specification.

Following is an example:

```
void __cdecl foo(int);  // C linkage with name mangled
void __cdecl foo(char)  // overload foo() with char is OK

void foo(int(*)());
   // overload on linkage of function
void foo(int (__cdecl *)());
   //pointer parameter is OK
extern "C++" {
   void __cdecl foo(int);
   // foo() has C linkage with name mangled
}

extern "C" {
   void __cdecl foo(int);
   // foo() has C linkage with name mangled
}
```

If the function is redeclared, the linkage keyword must appear in the first declaration; otherwise an error message is issued. Following are two examples:

```
int c_cf();
int __cdecl c_cf();
// Error 1251. The previous declaration did not have a linkage
specification
int __cdecl c_cf();
int c_cf();
  // OK, the linkage is inherited from the first declaration
```

Example of __cdecl use

The following example illustrates how you can use __cdecl to pass in a C parameter list from C++ code to a C function:

```
/*--------------------------------------------------------------------*/
/*  C++ source file                                                   */
/*--------------------------------------------------------------------*/
//
//  C++ Application:  passing a C++ function pointer to a C function
//
#include <stdio.h>

// C++ function declared with C calling convention
 void __cdecl callcxx() {
   printf(" I am a C++ function\n");
 }

// declare a function pointer with __cdecl linkage
 void (__cdecl *p1)();

// declare an extern C function,
// accepting a__cdecl function pointer
 extern "C" {
   void CALLC(void (__cdecl *pp)());
 }

// assign the function pointer to a __cdecl function
 int main() {
   p1 =  callcxx;

// call the C function with the __cdecl function pointer
   CALLC(p1);
 }
/*--------------------------------------------------------------------*/
/*  C source file                                                     */
/*--------------------------------------------------------------------*/
```

```
/*                                                                  */
/*  C Routine:  receiving a function pointer with C linkage         */
/*                                                                  */
#include <stdio.h>
extern void CALLC(void (*pp)()){
   printf(" I am a C function\n");
   (*pp)();   // call the function passed in
 }
```

Related reference:

"Language linkage (C++ only)" on page 9

The _Export function specifier (C++ only)

Use the `_Export` keyword with a function name to declare that it is to be exported (made available to other modules). You must define the function in the same translation unit in which you use the `_Export` keyword. For example:

```
int _Export anthony(float);
```

The above statement exports the function `anthony`, if you define the function within the translation unit.

The `_Export` keyword must immediately precede the function name. If the `_Export` keyword is repeated in a declaration, z/OS XL C++ issues a warning when you specify the INFO(GEN) option.

If you apply the `_Export` keyword to a class, the z/OS XL C++ compiler automatically exports all members of the class, whether `static`, `public`, `private`, or `protected`. However, if you want it to apply to individual class members, then you must apply it to each member that can be referenced. The following class definitions demonstrate this.

```
class A {
  public:
    int iii() {
      printf("Hi from A::iii()\n");
      aaa();
      printf("Call to A::ccc() returned %c\n", ccc());
      return 88;
    }
    static void _Export sss();
  protected:
    void _Export aaa();
  private:
    char _Export ccc();
};

class _Export B {
  public:
    int iii() {
      printf("Hi from B::iii()\n");
      aaa();
      printf("Call to B::ccc() returned %c\n", ccc());
      return 99;
    }
    static void sss();
  protected:
    void _Export aaa();
  private:
    char _Export ccc();
};
```

In the example below, both `X::Print()` and `X::GetNext()` will be exported.

```
class _Export X {
  public:
        ...
        void static Print();
        int GetNext();
        ...
};

void X:: static Print() {
        ...
}
int X::GetNext() {
        ...
}
```

The above examples demonstrate that you can either export specific members of a class or the entire class itself. Note that the `_Export` keyword can be applied to class tags in nested class declarations.

Related reference:

"External linkage" on page 8

"#pragma export" on page 553

Function return type specifiers

The result of a function is called its *return value* and the data type of the return value is called the *return type*.

C++ Every function declaration and definition must specify a return type, whether or not it actually returns a value. C++

C If a function declaration does not specify a return type, the compiler assumes an implicit return type of `int`. However, for conformance to C99, you should specify a return type for every function declaration and definition, whether or not the function returns `int`. C

A function may be defined 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. When a function does not return a value, `void` is the type specifier in the function declaration and definition.

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.

A function can have a return type that is a user-defined type. For example:

```
enum count {one, two, three};
enum count counter();
```

C The user-defined type may also be defined within the function declaration. C C++ The user-defined type may not be defined within the function declaration. C++

```
enum count{one, two, three} counter();   // legal in C
enum count{one, two, three} counter();   // error in C++
```

C++ References can also be used as return types for functions. The reference returns the lvalue of the object to which it refers. C++

Related reference:

"Type specifiers" on page 54

Function return values

C If a function is defined as having a return type of `void`, it should not return a value. C C++ In C++, a function which is defined as having a return type of `void`, or is a constructor or destructor, *must* not return a value. C++

C If a function is defined as having a return type other than `void`, it should return a value. Under compilation for strict C99 conformance, a function defined with a return type *must* include an expression containing the value to be returned. C

C++ A function defined with a return type *must* include an expression containing the value to be returned. C++

When a function returns a value, the value is returned via a `return` statement to the caller of the function, after being implicitly converted to the return type of the function in which it is defined. 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. C++

Related reference:

"The return statement" on page 212

"Overloading assignments" on page 334

"Overloading subscripting" on page 336

"The auto storage class specifier" on page 49

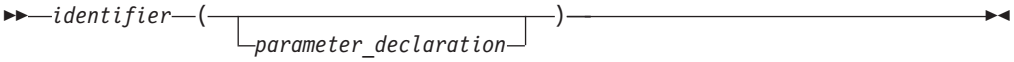
Function declarators

Function declarators consist of the following elements:

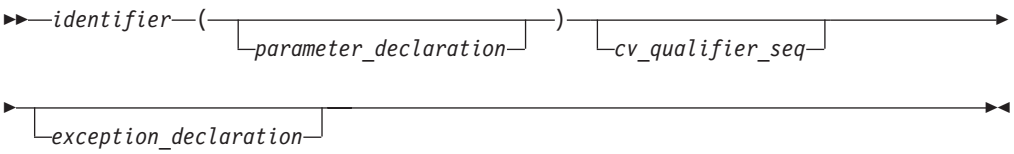
- An *identifier*, or name
- "Parameter declarations" on page 236, which specify the parameters that can be passed to the function in a function call.

- C++ Exception declarations, which include `throw` expressions; exception specifications are described in Chapter 16, "Exception handling (C++ only)," on page 485. C++
- C++ A *cv_qualifier_seq*, which represents one or a combination of `const` and `volatile`, and can be used only in class member functions. For the details of constant and volatile member functions, see "Constant and volatile member functions" on page 360. C++

Function declarator syntax (C only)



Function declarator syntax (C++ only)



Note: More complex types might be formed by using the syntax of *direct_declarator* in place of *identifier*. For the details of *direct_declarator*, see "Overview of declarators" on page 97.

Related reference:

"Default arguments in C++ functions (C++ only)" on page 254

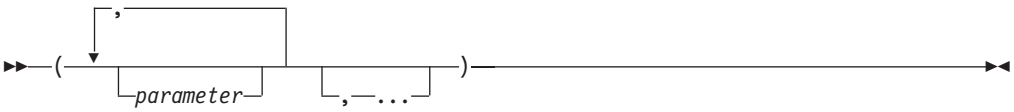
Chapter 4, "Declarators," on page 97

Parameter declarations

The function declarator includes the list of parameters that can be passed to the function when it is called by another function, or by itself.

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. C++

Function parameter declaration syntax



parameter

C++ An empty argument list in a function declaration or definition indicates a function that takes no arguments. To explicitly indicate that a function does not take any arguments, you can declare the function in two ways: with an empty parameter list, or with the keyword `void`:

```
int f(void);
int f();
```

C++

C An empty argument list in a function *definition* indicates that a function that takes no arguments. An empty argument list in a function *declaration* indicates that a function may take any number or type of arguments. Thus,

```
int f()
{
...
}
```

indicates that function `f` takes no arguments. However,

```
int f();
```

simply indicates that the number and type of parameters is not known. To explicitly indicate that a function does not take any arguments, you can replace the argument list with the keyword `void`.

```
int f(void);
```

C

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.

```
int f(int, ...);
```

C++ The comma before the ellipsis is optional. In addition, a parameter declaration is not required before the ellipsis. C++

C At least one parameter declaration, as well as a comma before the ellipsis, are both required in C. C

Parameter types

In a function *declaration*, or prototype, the type of each parameter must be specified. C++ In the function *definition*, the type of each parameter must also be specified. C++ C In the function *definition*, if the type of a parameter is not specified, it is assumed to be `int`. C

A variable of a user-defined type may be declared in a parameter declaration, as in the following example, in which `x` is declared for the first time:

```
struct X { int i; };
void print(struct X x);
```

C The user-defined type can also be defined within the parameter declaration. C C++ The user-defined type can not be defined within the parameter declaration. C++

```
void print(struct X { int i; } x);   // legal in C
void print(struct X { int i; } x);   // error in C++
```

Parameter names

In a function *definition*, each parameter must have an identifier. In a function *declaration*, or prototype, specifying an identifier is optional. Thus, the following example is legal in a function declaration:

```
int func(int,long);
```

C++ The following constraints apply to the use of parameter names in function declarations:

- Two parameters cannot have the same name within a single declaration.
- If a parameter name is the same as a name outside the function, 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. The first parameter name `subway_line` hides the namespace scope `enum` type and cannot be used again in the third parameter.

  ```
  enum subway_line {yonge, university, spadina, bloor};
  int subway(char * subway_line, int stations, subway_line intersects);
  ```

C++

Static array indices in function parameter declarations (C only)

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. An array type in the parameter list of a function is also 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.

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.

Note: This feature is C99 specific.

The following examples show how the feature can 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                 */
```

Related reference:

"The static storage class specifier" on page 49

"Arrays" on page 104

"Array subscripting operator [ ]" on page 171

"The void type" on page 59

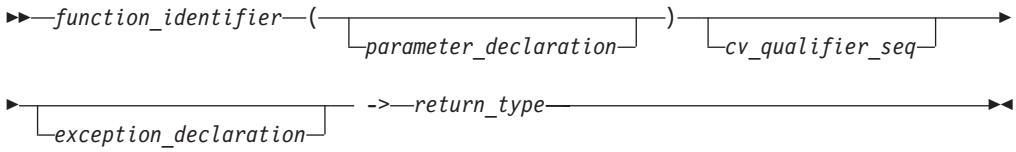
"Type specifiers" on page 54
"Type qualifiers" on page 85
"Exception specifications" on page 498

Trailing return type (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

The trailing return type feature removes a C++ limitation where the return type of a function template cannot be generalized if the return type depends on the types of the function arguments. For example, `a` and `b` are arguments of a function template `multiply(const A &a, const B &b)`, where `a` and `b` are of arbitrary types. Without the trailing return type feature, you cannot declare a return type for the `multiply` function template to generalize all the cases for `a*b`. With this feature, you can specify the return type after the function arguments. This resolves the scoping problem when the return type of a function template depends on the types of the function arguments.

Trailing return type syntax



Notes:

- This syntax is not the one for a function declaration or definition. The auto placeholder occurs in the syntax for declarations and definitions where they specify *return_type_specifier*.
- As with function declarators without a trailing return type, this syntax might be used to declare a pointer or reference to function.
- More complex types might be formed by using the syntax of *direct_declarator* in place of *function_identifier*. For the details of *direct_declarator*, see "Overview of declarators" on page 97.

To use the trailing return type feature, declare a generic return type with the `auto` keyword before the function identifier, and specify the exact return type after the function identifier. For example, use the `decltype` keyword to specify the exact return type.

In the following example, the `auto` keyword is put before the function identifier `add`. The return type of `add` is `decltype(a + b)`, which depends on the types of the function arguments `a` and `b`.

```
// Trailing return type is used to represent
// a fully generic return type for a+b.
template <typename FirstType, typename SecondType>
```

```
auto add(FirstType a, SecondType b) -> decltype(a + b){
   return a + b;
}

int main(){
   // The first template argument is of the integer type, and
   // the second template argument is of the character type.
   add(1, 'A');

   // Both the template arguments are of the integer type.
   add(3, 5);
}
```

Notes:

- When a trailing return type is used, the placeholder return type must be `auto`. For example, the statement `auto *f()->char` results in a compile-time error, because `auto *` is not allowed as the placeholder return type.
- The `auto` type specifier can be used with a function declarator with a trailing return type. Otherwise, the `auto` type specifier is used in accordance to the auto type deduction feature. For more information about auto type deduction, see "The auto type specifier (C++11)" on page 76. Because a function declaration cannot have an initializer as required for auto type deduction, the `auto` type specifier cannot be used in a function declaration without a trailing return type. For declarations of pointers and references to functions, the `auto` type specifier can be used with either a corresponding trailing return type or an initializer. For details of pointers and references to functions, see "Pointers to functions" on page 257.
- The return type of a function cannot be any of the following types:
  - Function
  - Array
  - Incomplete class
- The return type of a function cannot define any of the following types:
  - `struct`
  - `class`
  - `union`
  - `enum`

  However, the return type can be any of these types if the type is not defined in the function declaration.

In addition, this feature makes your program more compact and elegant in cases where functions have complicated return types. Without this feature, programs might be complicated and error prone. See the following example:

```
template <class A, class B> class K{
   public:
      int i;
};

K<int, double> (*(*bar())())() {
   return 0;
}
```

You can use the trailing return type feature to make the code compact. See the following example:

```
template <class A, class B> class K{
   public:
      int i;
```

```
};

auto bar()->auto(*)()->K<int, double>(*)(){
   return 0;
}
```

This feature can also be used for member functions of classes. In the following example, the program is concise because the return type of the member function `bar` does not need to be qualified after using a trailing return type:

```
struct A{
   typedef int ret_type;
   auto bar() -> ret_type;
};

// ret_type is not qualified
auto A::bar() -> ret_type{
   return 0;
}
```

Another use of this feature is in writing perfect forwarding functions. That is, the forwarding function calls another function, and the return type of the forwarding function is the same as that of the called function. See the following example:

```
double number (int a){
   return double(a);
}

int number(double b){
   return int(b);
}

template <class A>
auto wrapper(A a) -> decltype(number(a)){
   return number(a);
}

int main(){
   // The return value is 1.000000.
   wrapper(1);

   // The return value is 1.
   wrapper(1.5);
}
```

In this example, the `wrapper` function and the `number` function have the same return type.

Related reference:

"The auto type specifier (C++11)" on page 76

"The decltype(*expression*) type specifier (C++11)" on page 78

"C++11 compatibility" on page 640

"Member functions" on page 359

"Member functions of class templates" on page 445

"Function templates" on page 446

"Function declarations" on page 219

"Function definitions" on page 220

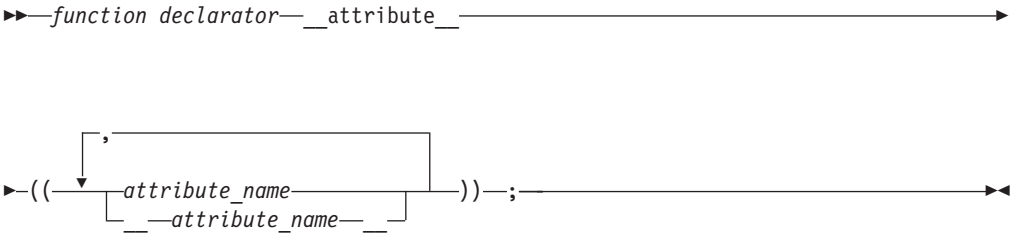
"Pointers to functions" on page 257

Function attributes (IBM extension)

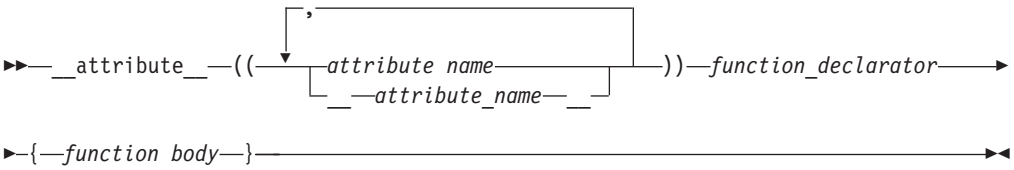
Function attributes are extensions implemented to enhance the portability of programs developed with GNU C. Specifiable attributes for functions provide explicit ways to help the compiler optimize function calls and to instruct it to check more aspects of the code. Others provide additional functionality.

A function attribute is specified with the keyword `__attribute__` followed by the attribute name and any additional arguments the attribute name requires. A function `__attribute__` specification is included in the declaration or definition of a function. The syntax takes the following forms:

Function attribute syntax: function declaration



Function attribute syntax: function definition (C only)



The function attribute in a function declaration is always placed after the declarator, including the parenthesized parameter declaration:

```
/* 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:

```
int __attribute__((individual_attribute_name)) foo(int i) { }
```

You can specify *attribute_name* with or without leading and trailing double underscore characters; however, using the double underscore characters reduces the likelihood of name conflicts with macros of the same name. These language features are collectively available when compiling in any of the extended language levels. C

The following function attributes are supported:

- "always_inline" on page 243
- "amode31 | amode64 (C only)" on page 243
- "armode | noarmode (C only)" on page 244

- "gnu_inline" on page 244
- "malloc" on page 245
- "used" on page 246

Related reference:

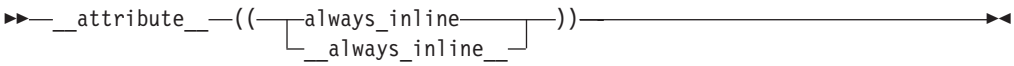
"Variable attributes (IBM extension)" on page 125

always_inline

The `always_inline` function attribute instructs the compiler to inline a function. This function can be inlined when all of the following conditions are satisfied:

- The function is an inline function that satisfies any of the following conditions:
  - The function is specified with the `inline` or `__inline__` keyword.
  - C++ The function is defined within a class declaration. C++
- The function is not specified with the `noinline` or `__noinline__` attribute.
- C The optimization level is at OPTIMIZE(2) or higher. C
- The number of functions to be inlined does not exceed the limit of inline functions that can be supported by the compiler.

always_inline function attribute syntax



The `noinline` attribute takes precedence over the `always_inline` attribute. The `always_inline` attribute takes precedence over inlining compiler options only if inlining is enabled. The `always_inline` attribute is ignored if inlining is disabled.

C++ The compiler might not inline a virtual function even when the function is specified with the `always_inline` attribute. The compiler will not issue an informational message to indicate that a virtual function is not inlined.

When you specialize a function template that is specified with the `always_inline` attribute, this attribute is propagated to the template specification. If you apply the `always_inline` attribute to the template specification, the duplicate `always_inline` attribute is ignored. See the following example.

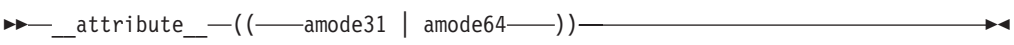
```
template<class T> inline __attribute__((always_inline)) T test( ){
    return (T)0;
}

// The duplicate attribute "always_inline" is ignored.
template<> inline __attribute__((always_inline)) float test<float>(){
    return (float)0;
}
```

C++

amode31 | amode64 (C only)

When the METAL compiler option is in effect, it is possible for the application to switch between 31-bit and 64-bit addressing modes during function calls and returns. You can use the `amode31` or `amode64` function attribute to identify the addressing mode of the called function.

amode31 | amode64 function attribute syntax



The following example declares the function `foo` to be in AMODE 64 mode:

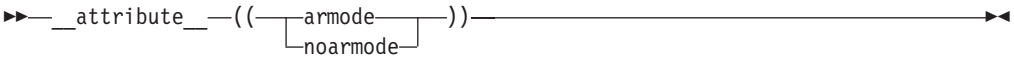
```
void foo() __attribute__ ((amode64));
```

For more information on the METAL compiler option, see the METAL compiler option description in *z/OS XL C/C++ User's Guide*. For more information on AMODE switching, see *z/OS Metal C Programming Guide and Reference.*

armode | noarmode (C only)

When the METAL compiler option is in effect, you can use the `armode` function attribute to specify whether or not a given function is to operate in access-register (AR) mode. AR mode allows a C function to access multiple additional data spaces, and manipulate more data in memory.

armode function attribute syntax



Functions in AR mode can call functions not in AR mode, and vice versa.

The following example declares the function `foo` to be in AR mode:

```
void foo() __attribute__((armode));
```

The attribute overrides the default setting of the ARMODE compiler option for the specified function. Note that this attribute is only supported when the METAL compiler option is in effect.

For more information on ARMODE and METAL compiler options, see ARMODE and METAL compiler options in the *z/OS XL C/C++ User's Guide*.

Related reference:

"The armode | noarmode type attribute (C only)" on page 94

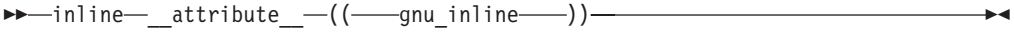
"The __far type qualifier (C only)" on page 88

gnu_inline

The `gnu_inline` attribute instructs the compiler to modify the inlining behavior of a function. When this function attribute is used, the compiler imitates the GNU legacy inlining extension to C.

This function attribute is only enabled if used in conjunction with an inline keyword (`__inline__`, `inline`, `__inline`, etc.).

gnu_inline function attribute syntax



Note: The behavior of the `gnu_inline` function attribute is the same when used in conjunction with either the `inline` or `__inline__` keywords.

The semantics of the GNU legacy inlining extension to C are as follows:

C

extern gnu_inline:

```
extern inline __attribute__((gnu_inline)) func() {...};
```

This definition of `func` is used only for inlining. It is not compiled as a standalone function.

static gnu_inline:

```
static inline __attribute__((gnu_inline)) func() {...};
```

If the function is generated, it is generated with internal linkage.

plain gnu_inline:

```
inline __attribute__((gnu_inline)) func() {...};
```

The definition is used for inlining when possible. It is compiled as a standalone function (emitted as a strong definition) and emitted with external linkage.

C

C++

extern gnu_inline:

```
[extern] inline __attribute__((gnu_inline)) func() {...};
```

This definition of `func` is used only for inlining. It is not compiled as a standalone function. Note that member functions (including static ones and ones with no linkage) marked with function attribute `gnu_inline` has "extern" behavior.

static gnu_inline:

```
static inline __attribute__((gnu_inline)) func() {...};
```

If the function is generated, it is generated with internal linkage. Note that static behavior only applies to non-member static functions.

C++

The `gnu_inline` attribute can be specified inside double parentheses with keyword `__attribute__` in a function declaration. See the following example.

```
inline int func() __attribute__((gnu_inline));
```

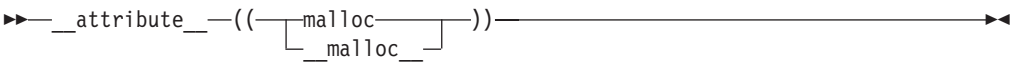
As with other GCC function attributes, the double underscores on the attribute name are optional. The `gnu_inline` attribute should be used with a function that is also declared with the `inline` keyword.

malloc

With the function attribute `malloc`, you can instruct the compiler to treat a function as if any non-NULL pointer it returns cannot alias any other valid pointers. This type of function (such as `malloc` and `calloc`) has this property, hence the name of the attribute. As with all supported attributes, `malloc` will be accepted by the compiler without requiring any particular option or language level.

The malloc function attribute can be specified inside double parentheses via keyword __attribute__ in a function declaration.

malloc function attribute syntax



As with other GCC function attributes, the double underscores on the attribute name are optional.

Note:

- Do not use this function attribute unless you are sure that the pointer returned by a function points to unique storage. Otherwise, optimizations performed might lead to incorrect behavior at run time.
- If the function does not return a pointer or C++ reference return type but it is marked with the function attribute malloc, a warning is emitted, and the attribute is ignored.

Example

A simple case that should be optimized when attribute malloc is used:

```
#include <stdlib.h>
#include <stdio.h>
int a;
void* my_malloc(int size)  __attribute__ ((__malloc__))
{
  void* p = malloc(size);
  if (!p) {
    printf("my_malloc: out of memory!\n");
    exit(1);
  }
  return p;
}
int main() {
  int* x = &a;
  int* p = (int*) my_malloc(sizeof(int));
  *x = 0;
  *p = 1;
  if (*x) printf("This printf statement to be detected as unreachable
                  and discarded during compilation process\n");
  return 0;
}
```

used

When a function is referenced only in inline assembly, you can use the used function attribute to instruct the compiler to emit the code for the function even if it appears that the function is not referenced.

The used function attribute can be specified inside double parentheses via keyword __attribute__ in a function declaration, for example, `int foo() __attribute__ ((__used__));` As with other GCC function attributes, the double underscores on the attribute name are optional.

used function attribute syntax

If the function attribute gnu_inline is specified in such a way that the function is discarded, and is specified together with the function attribute used, the gnu_inline attribute wins, and the function definition is discarded.

The main() function

When a program begins running, the system calls the function main, which marks the entry point of the program. By default, main has the storage class extern. Every program must have one function named main, and the following constraints apply:

- No other function in the program can be called main.
- main cannot be defined as inline or static.
- C++ main cannot be called from within a program. C++
- C++ The address of main cannot be taken. C++
- C++ The main function cannot be overloaded. C++
- C++11 The main function cannot be declared with the constexpr specifier. C++11

The function main can be defined with or without parameters, using any of the following forms:

```
int main (void){}
int main ( ){}
int main(int argc, char *argv[]){}
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) is an integer that indicates how many arguments were entered on the command line when the program was started. The second parameter, argv (argument vector), is an array of pointers to arrays of character objects. The array objects are null-terminated strings, representing the arguments that were entered on the command line when the program was started.

The first element of the array, argv[0], is a pointer to the character array that contains the program name or invocation name of the program that is being run from the command line. argv[1] indicates the first argument passed to the program, argv[2] the second argument, and so on.

The following example 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]);
  printf("\n");
}
```

Invoking this program from a command line:

```
backward string1 string2
```

gives the following output:

```
string2 string1
```

The arguments `argc` and `argv` would contain the following values at the start of the program:

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: See *z/OS XL C/C++ Programming Guide* for details about receiving the parameter list (`argv`) in C `main`, preparing your **`main`** function to receive parameters, and on C and C++ parameter passing considerations.

Related reference:

"The extern storage class specifier" on page 51

"The inline function specifier" on page 228

"The static storage class specifier" on page 49

"Function calls" on page 249

Command-line arguments

z/OS XL C/C++ treats arguments that you enter on the command line differently in different environments. The following lists how `argv` and `argc` are handled.

The maximum allowable length of a command-line argument for z/OS Language Environment is 64K.

Under z/OS batch

argc Returns the number of strings in the argument line

argv[0]
Returns the program name in uppercase

argv[*1 to n***]**
Returns the arguments as you enter them

Under IBM IMS™

argc Returns 1

argv[0]
Is a null pointer

Under IBM CICS®

argc Returns 1

argv[0]
Returns the transaction ID

Under TSO command

argc Returns the number of strings in the argument line

argv[0]
Returns the program name in uppercase

argv[*1 to n*]
Arguments entered in uppercase are returned in lowercase. Arguments entered in mixed or lowercase are returned as entered.

Under TSO call

Without the `ASIS` option:

argc Returns the number of strings in the argument line

argv Returns the program name and arguments in lowercase

With the `ASIS` option:

argc Returns the number of strings in the argument line

argv[0]
Returns the program name in uppercase

argv[*1 to n*]
Arguments entered in uppercase are returned in lowercase. Arguments entered in mixed or lowercase are returned as entered.

Under z/OS UNIX System Services shell

argc Returns the number of strings in the argument line

argv[0]
Returns the program name as you enter it

argv[*1 to n*]
Returns the arguments exactly as you enter them

The only delimiter for the arguments that are passed to `main()` is white space. z/OS XL C/C++ uses commas passed to `main()` by JCL as arguments and not as delimiters.

The following example appends the comma to the `'one'` when passed to `main()`.

```
//FUNC  EXEC PCGO,GPGM='FUNC',
//       PARM.GO=('one',
//       'two')
```

For more information on restrictions of the command-line arguments, refer to *z/OS XL C/C++ User's Guide*.

Related reference:

"Function calls"

"Type specifiers" on page 54

"Identifiers" on page 16

"Block statements" on page 199

Function calls

After a function is declared and defined, it can be *called* from anywhere within the program: from within the `main` function, from another function, and even from itself. Calling the function involves specifying the function name, followed by the function call operator and any data values the function expects to receive. These values are the *arguments* for the parameters defined for the function. This process is called *passing arguments* to the function.

You can pass arguments to the called functions in three ways:

- "Pass by value," which copies the *value* of an argument to the corresponding parameter in the called function;
- "Pass by pointer" on page 251, which passes a *pointer* argument to the corresponding parameter in the called function;
- C++ "Pass by reference (C++ only)" on page 252, which passes the *reference* of an argument to the corresponding parameter in the called function. C++

C++ 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 object that is actually passed by reference.

The compiler generates an error when a function argument is a class object and all of the following conditions are true:

- 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.

C++

C A function call is always an rvalue. C

C++ A function call belongs to one of the following value categories depending on the result type of the function:

- An lvalue if the result type is an lvalue reference type C++11 or an rvalue reference to a function type C++11
- C++11 An xvalue if the result type is an rvalue reference to an object type C++11
- A C++11 (prvalue) C++11 rvalue in other cases

C++

Related reference:

"Command-line arguments" on page 248

"Function argument conversions" on page 139

"Function call expressions" on page 149

"Constructors" on page 408

"Lvalues and rvalues" on page 141

"References (C++ only)" on page 107

Pass by value

When you use pass-by-*value*, the compiler copies the value of an argument in a calling function to a corresponding non-pointer or non-reference parameter in the called function definition. The parameter in the called function is initialized with the value of the passed argument. As long as the parameter has not been declared as constant, the value of the parameter can be changed, but the changes are only performed within the scope of the called function only; they have no effect on the value of the argument in the calling function.

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.

```
/**
 ** This example illustrates calling a function by value
 **/

#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;
}
```

The output of the program is:

```
In func, a = 12 b = 7
In main, x = 5  y = 7
```

Pass by pointer

Pass-by-pointer means to pass a pointer argument in the calling function to the corresponding formal parameter of the called function. The called function can modify the value of the variable to which the pointer argument points.

The following example shows how arguments are passed by pointer:

```
#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 values of the variables `a` and `b` are exchanged because they are passed by pointer. The output is:

```
A is 20 and B is 10
```

When you use pass-by-pointer, a copy of the pointer is passed to the function. If you modify the pointer inside the called function, you only modify the copy of the pointer, but the original pointer remains unmodified and still points to the original variable.

The difference between pass-by-pointer and pass-by-value is that modifications made to arguments passed in by pointer in the called function have effect in the calling function, whereas modifications made to arguments passed in by value in

the called function can not affect the calling function. Use pass-by-pointer if you want to modify the argument value in the calling function. Otherwise, use pass-by-value to pass arguments.

Related reference:

"Pointers" on page 100

Pass by reference (C++ only)

Pass-by-reference means to pass the reference of an argument in the calling function to the corresponding formal parameter of the called function. The called function can modify the value of the argument by using its reference passed in.

The following example shows how arguments are passed by reference. The reference parameters are initialized with the actual arguments when the function is called.

CCNX06A

```
#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 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
```

To modify a reference that is qualified by the `const` qualifier, you must cast away its constness with the `const_cast` operator. For example:

```
#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`.

Pass-by-references is more efficient than pass-by-value, because it does not copy the arguments. The formal parameter is an alias for the argument. When the called function read or write the formal parameter, it is actually read or write the argument itself.

The difference between pass-by-reference and pass-by-value is that modifications made to arguments passed in by reference in the called function have effect in the calling function, whereas modifications made to arguments passed in by value in the called function can not affect the calling function. Use pass-by-reference if you want to modify the argument value in the calling function. Otherwise, use pass-by-value to pass arguments.

The difference between pass-by-reference and pass-by-pointer is that pointers can be `NULL` or reassigned whereas references cannot. Use pass-by-pointer if `NULL` is a valid parameter value or if you want to reassign the pointer. Otherwise, use constant or non-constant references to pass arguments.

Related reference:

"References (C++ only)" on page 107

"The const_cast operator (C++ only)" on page 181

Allocation and deallocation functions (C++ only)

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;
}
```

See the following 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 reference:

"new expressions (C++ only)" on page 185

Default arguments in C++ functions (C++ only)

You can provide default values for function parameters. For example:

CCNX06B

```
#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.
```

In this example, 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) );
```

Related reference:

"Pointers to functions" on page 257

Restrictions on default arguments (C++ only)

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 reference:

"Function call expressions" on page 149

"new expressions (C++ only)" on page 185

Evaluation of default arguments (C++ only)

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;
};
```

Pointers to functions

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.

For z/OS XL C/C++, use the `__cdecl` keyword to declare a pointer to a function as a C linkage. For more information, refer to "The __cdecl function specifier (C++ only)" on page 231.

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. Function parameters have precedence over pointers in declarations, so parentheses are required to alter the precedence and declare a pointer to a function. Without them, the compiler interprets the declaration 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        */
```

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`.

C++11

You can use a trailing return type in the declaration or definition of a pointer to a function. For example:

```
auto(*fp)()->int;
```

In this example, `fp` is a pointer to a function that returns `int`. You can rewrite the declaration of `fp` without using a trailing return type as `int (*fp)(void)`. For more information on trailing return type, see "Trailing return type (C++11)" on page 239.

C++11

Under z/OS XL C/C++, if you pass a function pointer to a function, or the function returns a function pointer, the declared or implied linkages must be the same. Use the `extern` keyword with declarations in order to specify different linkages.

The following example illustrates the correct and incorrect uses of function pointers under z/OS XL C/C++ :

```
#include <stdlib.h>

extern "C"   int cf();
extern "C++" int cxxf(); //  C++ is included here for clarity;
                         //  it is not required; if it is
                         //  omitted, cxxf() will still have
                         //  C++ linkage.
extern "C"   int (*c_fp)();
extern "C++" int (*cxx_fp)();
typedef int (*dft_fp_T)();
typedef int (dft_f_T)();
```

```
extern "C" {
  typedef void (*cfp_T)();
  typedef int (*cf_pT)();
  void cfn();
  void (*cfp)();
}

extern "C++" {
  typedef int (*cxxf_pT)();
  void cxxfn();
  void (*cxxfp)();
}

extern "C" void f_cprm(int (*f)()) {
  int (*s)() = cxxf;     //  error, incompatible linkages-cxxf has
                         //  C++ linkage, s has C linkage as it
                         //  is included in the extern "C" wrapper
  cxxf_pT j  = cxxf;     //  valid, both have C++ linkage
  int (*i)() = cf;       //  valid, both have C linkage

}

extern "C++" void f_cxprm(int (*f)()) {
  int (*s)() = cf;       //  error, incompatible linkages-cf has C
                         //  linkage, s has C++ linkage as it is
                         //  included in the extern "C++" wrapper
  int (*i)() = cxxf;     //  valid, both have C++ linkage
  cf_pT   j  = cf;       //  valid, both have C linkage
}
main() {

  c_fp    = cxxf;           //  error - c_fp has C linkage and cxxf has
                            //  C++ linkage
  cxx_fp  = cf;             //  error - cxx_fp has C++ linkage and
                            //  cf has C linkage
  dft_fp_T dftfpT1 = cf;    //  error - dftfpT1 has C++ linkage and
                            //  cf has C linkage
  dft_f_T  *dftfT3 = cf;    //  error - dftfT3 has C++ linkage and
                            //  cf has C linkage
  dft_fp_T dftfpT5 = cxxf;  //  valid
  dft_f_T  *dftfT6 = cxxf;  //  valid

  c_fp    = cf;             //  valid
  cxx_fp  = cxxf;           //  valid
  f_cprm(cf);               //  valid
  f_cxprm(cxxf);            //  valid

  // The following errors are due to incompatible linkage of function
  // arguments, type conversion not possible
  f_cprm(cxxf);             //  error - f_cprm expects a parameter with
                            //  C linkage, but cxxf has C++ linkage
  f_cxprm(cf);              //  error - f_cxprm expects a parameter
                            //  with C++ linkage, but cf has C linkage
}
```

For z/OS, linkage compatibility affects all C library functions that accept a function pointer as a parameter.

References to functions

A reference to a function is an alias or an alternative name for a function. You must initialize all references to functions after they are defined. Once defined, a reference to a function cannot be reassigned. You can use references to call functions and to pass functions as arguments to other functions. For example:

```
int g();

// f is a reference to a function that has no parameters and returns int.
int bar(int(&f)()){

   // call function f that is passed as an argument.
   return f();
}

int x = bar(g);
```

C++11

You can also use a trailing return type in the declaration or definition of a reference to a function. In the following example, `fp` is a reference to a function that returns `int`. For more information on trailing return type, see "Trailing return type (C++11)" on page 239.

```
auto(&fp)()->int;
```

C++11

Related reference:

"Language linkage (C++ only)" on page 9

"Pointers" on page 100

"Pointer conversions" on page 137

"References (C++ only)" on page 107

"The extern storage class specifier" on page 226

Atomic library (C11)

The header file `stdatomic.h` defines the following macros, types, and functions for performing atomic operations on data shared among threads:

- "Macros (C11)"
- "Types (C11)" on page 260
- "Atomic library functions (C11)" on page 262

Macros (C11)

Atomic lock-free macros

The following atomic lock-free macros indicate the lock-free property of the corresponding atomic types (both signed and unsigned).

- `#define ATOMIC_BOOL_LOCK_FREE` *value*
- `#define ATOMIC_CHAR_LOCK_FREE` *value*
- `#define ATOMIC_CHAR16_T_LOCK_FREE` *value*
- `#define ATOMIC_CHAR32_T_LOCK_FREE` *value*
- `#define ATOMIC_WCHAR_T_LOCK_FREE` *value*
- `#define ATOMIC_SHORT_LOCK_FREE` *value*
- `#define ATOMIC_INT_LOCK_FREE` *value*
- `#define ATOMIC_LONG_LOCK_FREE` *value*
- `#define ATOMIC_LLONG_LOCK_FREE` *value*
- `#define ATOMIC_POINTER_LOCK_FREE` *value*

The values of the atomic lock-free macros are defined as follows:

0 Indicates that the atomic type is never lock-free.

1 Indicates that the atomic type is sometimes lock-free.

2 Indicates that the atomic type is always lock-free.

The ATOMIC_FLAG_INIT macro

The following macro expands to an expression that you can use to initialize an object of type atomic_flag.

```
#define ATOMIC_FLAG_INIT value
```

If the value of atomic_flag is not initialized with this macro, it is undefined.

The ATOMIC_VAR_INIT macro

The following macro expands to an expression that you can use to initialize an atomic variable of a type that is initialization-compatible with *value*. If an atomic object of automatic storage duration is not initialized with this macro, its initial value is undefined.

```
#define ATOMIC_VAR_INIT(value)
```

If an atomic object with automatic storage duration is not explicitly initialized with ATOMIC_VAR_INIT, it is initially in an indeterminate state. However, the default initialization for objects with static or thread-local storage duration produces a valid state. Concurrent accesses to the variable that is being initialized, even by an atomic operation, constitute a data race.

The kill_dependency macro

The following macro ends a dependency chain. That is, the argument does not carry a dependency to the return value.

```
type kill_dependency(type y);
```

The kill_dependency macro returns the value of y.

Types (C11)

The <stdatomic.h> header file defines two types, memory_order and atomic_flag.

The memory_order type

The enumerated type memory_order determines how non-atomic memory accesses are ordered around an atomic operation.

```
enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
};
```

Its enumeration constants define memory order constraints as follows:

memory_order_relaxed
There are no constraints on reordering of memory accesses around the atomic variable.

memory_order_consume
In the current thread, no reads that are dependent on the currently loaded value can be reordered before this load.

memory_order_acquire
In the current thread, no reads can be reordered before this load.

memory_order_release
In the current thread, no writes can be reordered after this store.

memory_order_acq_rel
In the current thread, no reads can be reordered before this load, and that no writes can be reordered after this store.

memory_order_seq_cst
The operation has the same semantics as the acquire-release operation. In addition, it has a sequentially consistent operation order.

The atomic_flag type

atomic_flag is a structure type that represents a lock-free, primitive atomic flag, and several atomic analogs of integer types. It provides the classic test-and-set function, and has two states: set and clear.

Operations on an object of type atomic_flag must be lock-free.

You can use the macro ATOMIC_FLAG_INIT to initialize an atomic_flag object to the clear state. If an atomic_flag object is not explicitly initialized with ATOMIC_FLAG_INIT, it is initially in an indeterminate state.

Atomic integer types (C11)

In the following table, the atomic type name is declared as a type that has the same representation and alignment requirements as its corresponding direct type. You can use an atomic type and its corresponding direct type interchangeably as arguments to functions, return values from functions, or members of unions.

Table 33. Atomic type names and direct types

Atomic type name	Direct type
atomic_bool	_Atomic _Bool
atomic_char	_Atomic char
atomic_schar	_Atomic signed char
atomic_uchar	_Atomic unsigned char
atomic_short	_Atomic short
atomic_ushort	_Atomic unsigned short
atomic_int	_Atomic int
atomic_uint	_Atomic unsigned int
atomic_long	_Atomic long
atomic_ulong	_Atomic unsigned long
atomic_llong	_Atomic long long
atomic_ullong	_Atomic unsigned long long
atomic_char16_t	_Atomic char16_t
atomic_char32_t	_Atomic char32_t

Table 33. Atomic type names and direct types (continued)

Atomic type name	Direct type
atomic_wchar_t	_Atomic wchar_t
atomic_int_least8_t	_Atomic int_least8_t
atomic_uint_least8_t	_Atomic uint_least8_t
atomic_int_least16_t	_Atomic int_least16_t
atomic_uint_least16_t	_Atomic uint_least16_t
atomic_int_least32_t	_Atomic int_least32_t
atomic_uint_least32_t	_Atomic uint_least32_t
atomic_int_least64_t	_Atomic int_least64_t
atomic_uint_least64_t	_Atomic uint_least64_t
atomic_int_fast8_t	_Atomic int_fast8_t
atomic_uint_fast8_t	_Atomic uint_fast8_t
atomic_int_fast16_t	_Atomic int_fast16_t
atomic_uint_fast16_t	_Atomic uint_fast16_t
atomic_int_fast32_t	_Atomic int_fast32_t
atomic_uint_fast32_t	_Atomic uint_fast32_t
atomic_int_fast64_t	_Atomic int_fast64_t
atomic_uint_fast64_t	_Atomic uint_fast64_t
atomic_intptr_t	_Atomic intptr_t
atomic_uintptr_t	_Atomic uintptr_t
atomic_size_t	_Atomic size_t
atomic_ptrdiff_t	_Atomic ptrdiff_t
atomic_intmax_t	_Atomic intmax_t
atomic_uintmax_t	_Atomic uintmax_t

Atomic library functions (C11)

atomic_init (C11)

Purpose

Initializes an atomic object.

Prototype

void atomic_init(volatile *A* **object*, *C value*);

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
A pointer that points to the atomic object to initialize.

value
The value that is used to initialize the atomic object.

Usage

You can use this function to initialize an atomic object, but this function does not avoid data races. Concurrent accesses to the variable that is being initialized, even by an atomic operation, constitute a data race.

atomic_thread_fence (C11)
Purpose

Establishes a memory synchronization order of non-atomic and relaxed atomic accesses without an associated atomic operation.

Prototype

void atomic_thread_fence(memory_order *order*);

Parameter

order
The memory order that is executed by this fence.

Usage

The function has the following effects depending on the value of *order*:

- If the value of *order* is `memory_order_relaxed`, the function has no effects.
- If the value of *order* is `memory_order_acquire` or `memory_order_consume`, the function is an acquire fence.
- If the value of *order* is `memory_order_release`, the function is a release fence.
- If the value of *order* is `memory_order_acq_rel`, the function is both an acquire fence and a release fence.
- If the value of *order* is `memory_order_seq_cst`, the function is a sequentially consistent acquire and release fence.

atomic_signal_fence (C11)
Purpose

Establishes a memory synchronization order of non-atomic and relaxed atomic accesses between a thread and a signal handler that is executed on the same thread.

Prototype

void atomic_signal_fence(memory_order *order*);

Parameters

order
The memory order that is executed by this fence.

Usage

This function is equivalent to `atomic_thread_fence`, except that the resulting order constraints are established only between a thread and a signal handler that is executed in the same thread.

You can use `atomic_signal_fence` to specify the order in which operations performed by the thread become visible to the signal handler.

Memory is affected according to the value of *order*.

atomic_is_lock_free (C11)

Purpose

Indicates whether an atomic object is lock-free.

Prototype

_Bool atomic_is_lock_free(const volatile *A* **object*);

Parameters

A One of the atomic types.

object
A pointer that points to the atomic object to inspect.

Return value

Returns a nonzero value only if the operations of the object are lock-free. Otherwise, zero is returned.

atomic_store (C11)

Purpose

Stores a value in an atomic object.

Prototypes

void atomic_store(volatile *A* **object*, *C value*);

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
A pointer that points to the atomic object to modify.

value
The value to be stored in the atomic object.

Usage

You can use this function to atomically store *value* in the atomic object that *object* points to.

atomic_store_explicit (C11)
Purpose

Stores a value in an atomic object.

Prototype

void atomic_store_explicit(volatile *A* **object*, *C value*, memory_order *order*);

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
A pointer that points to the atomic object to modify.

value
The value to be stored in the atomic object.

order
The memory synchronization order for this operation.

Usage

The *order* argument cannot be `memory_order_acquire`, `memory_order_consume`, or `memory_order_acq_rel`.

You can use this function to atomically store *value* in the atomic object that *object* points to. Memory is affected according to the value of *order*.

atomic_load (C11)
Purpose

Reads a value from an atomic object.

Prototypes

C atomic_load(volatile *A* **object*);

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
A pointer that points to the atomic object to access.

Return value

Atomically returns the value pointed to by *object*.

atomic_load_explicit (C11)
Purpose

Reads a value from an atomic object.

Prototypes

C atomic_load_explicit(volatile *A* **object*, memory_order *order*);

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
A pointer that points to the atomic object to access.

order
The memory synchronization order for this operation. The *order* parameter cannot be `memory_order_release` or `memory_order_acq_rel`.

Return value

Atomically returns the value pointed to by *object*.

Memory is affected according to the value of *order*.

atomic_exchange (C11)
Purpose

Replaces the value of an atomic object with a specified value.

Prototypes

C atomic_exchange(volatile *A* **object*, *C value*);

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
A pointer that points to the atomic object to modify.

value
The value to replace the atomic object with.

Usage

You can atomically replace the value pointed to by *object* with *value*.

Return value

Atomically returns the value pointed to by *object* immediately before the exchange.

atomic_exchange_explicit (C11)
Purpose

Replaces the value of an atomic object with a specified value.

Prototypes

C atomic_exchange_explicit(volatile *A* **object*, *C value*, memory_order *order*);

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
A pointer that points to the atomic object to modify.

value
The value to replace the atomic object with.

order
The memory synchronization order for this operation.

Usage

This function atomically replaces the value pointed to by *object* with *value*. Memory is affected according to the value of *order*.

Return value

Atomically returns the value pointed to by *object* immediately before the exchange.

atomic_compare_exchange_strong (C11)
Purpose

Performs an atomic compare and exchange operation.

Prototypes

_Bool atomic_compare_exchange_strong(volatile *A* **object*, *C* **expected*, *C desired*);

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
A pointer that points to the atomic object to test and modify.

expected
A pointer that points to the value expected to be found in the atomic object.

desired
The value to be stored in the atomic object if the comparison result is true.

Usage

This function atomically compares the value pointed to by *object* for equality with the value pointed to by *expected*.

If the comparison result is true, this function replaces the value pointed to by *object* with *desired*.

If the comparison result is false, this function updates the value pointed to by *expected* with the value pointed to by *object*.

Return value

Returns the result of the comparison.

atomic_compare_exchange_strong_explicit (C11)

Purpose

Performs an atomic compare and exchange operation.

Prototypes

_Bool atomic_compare_exchange_strong_explicit(volatile *A* **object*, *C* **expected*, *C desired*, memory_order *success*, memory_order *failure*);

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
A pointer that points to the atomic object to test and modify.

expected
A pointer that points to the value expected to be found in the atomic object.

desired
The value to be stored in the atomic object if the comparison result is true.

success
The memory synchronization order for the read-modify-write operation if the comparison succeeds.

failure
The memory synchronization order for the load operation if the comparison fails. This parameter cannot be `memory_order_release` or `memory_order_acq_rel`. You cannot specify it with a memory synchronization order that is stronger than *success*.

Usage

This function atomically compares the value pointed to by *object* for equality with the value pointed to by *expected*.

If the comparison result is true, this function replaces the value pointed to by *object* with *desired*.

If the comparison result is false, this function updates the value pointed to by *expected* with the value pointed to by *object*.

If the comparison result is true, memory is affected according to the value of *success*, and if the comparison result is false, memory is affected according to the value of *failure*.

Return value

Returns the result of the comparison.

atomic_compare_exchange_weak (C11)
Purpose

Performs an atomic compare and exchange operation.

Prototypes

_Bool atomic_compare_exchange_weak(volatile *A* **object*, *C* **expected*, *C desired*);

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
A pointer that points to the atomic object to test and modify.

expected
A pointer that points to the value expected to be found in the atomic object.

desired
The value to be stored in the atomic object if the comparison result is true.

Usage

This function atomically compares the value pointed to by *object* for equality with the value pointed to by *expected*.

If the comparison result is true, this function replaces the value pointed to by *object* with *desired*.

If the comparison result is false, this function updates the value pointed to by *expected* with the value pointed to by *object*.

A weak compare-and-exchange operation might fail spuriously. That is, even when the contents of memory pointed to by *expected* and *object* are equal, it might return zero and store back to the place pointed to by *expected* the same memory contents that were originally there.

Return value

Returns the result of the comparison.

atomic_compare_exchange_weak_explicit (C11)
Purpose

Performs an atomic compare and exchange operation.

Prototypes

_Bool atomic_compare_exchange_weak_explicit(volatile *A* **object*, *C* **expected*, *C desired*, memory_order *success*, memory_order *failure*);

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
A pointer that points to the atomic object to test and modify.

expected
A pointer that points to the value expected to be found in the atomic object.

desired
The value to be stored in the atomic object if the comparison result is true.

success
The memory synchronization order for the read-modify-write operation if the comparison succeeds.

failure
The memory synchronization order for the load operation if the comparison fails. This parameter cannot be `memory_order_release` or `memory_order_acq_rel`. You cannot specify it with a memory synchronization order stronger than *success*.

Usage

This function atomically compares the value pointed to by *object* for equality with the value pointed to by *expected*.

If the comparison result is true, this function replaces the value pointed to by *object* with *desired*.

If the comparison result is false, this function updates the value pointed to by *expected* with the value pointed to by *object*.

If the comparison is true, memory is affected according to the value of *success*, and if the comparison is false, memory is affected according to the value of *failure*.

A weak compare-and-exchange operation might fail spuriously. That is, even when the contents of memory referred to by *expected* and *object* are equal, it might return zero and store back to the place pointed to by *expected* the same memory contents that were originally there.

Return value

Returns the result of the comparison.

atomic_fetch_add (C11)
Purpose

Performs the addition operation on an atomic object.

Prototypes

C atomic_fetch_add(volatile *A* **object*, *M operand*);

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
A pointer that points to the atomic object to modify.

M The type of a specified operand is determined as follows:
- If *A* is an atomic integer type, *M* is the corresponding non-atomic type of *A*.
- If *A* is an atomic pointer type, *M* is the `ptrdiff_t` type.

operand
A specified value to add to the value in the atomic object.

Usage

This function adds the value pointed to by *object* and the specified *operand*, and stores the addition result in the atomic object pointed to by *object*.

Return value

Returns atomically the value pointed to by *object* before the addition.

atomic_fetch_add_explicit (C11)
Purpose

Performs the addition operation on an atomic object.

Prototypes

C atomic_fetch_add_explicit(volatile *A* **object*, *M operand*, memory_order *order*);

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
A pointer that points to the atomic object to modify.

M The type of a specified operand is determined as follows:
- If *A* is an atomic integer type, *M* is the corresponding non-atomic type of *A*.
- If *A* is an atomic pointer type, *M* is the `ptrdiff_t` type.

operand
A specified value to add to the value in the atomic object.

order
The memory synchronization order for this operation.

Usage

This function adds the value pointed to by *object* and the specified *operand*, and stores the addition result in the atomic object pointed to by *object*.

Memory is affected according to the value of *order*.

Return value

Returns atomically the value pointed to by *object* before the addition.

atomic_fetch_sub (C11)
Purpose

Performs the subtraction operation on an atomic object.

Prototypes

C atomic_fetch_sub(volatile *A* **object*, *M operand*);

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
A pointer that points to the atomic object to modify.

M The type of a specified operand is determined as follows:
- If *A* is an atomic integer type, *M* is the corresponding non-atomic type of *A*.
- If *A* is an atomic pointer type, *M* is the `ptrdiff_t` type.

operand
A specified value to subtract from the value in the atomic object.

Usage

This function subtracts the specified *operand* from the value pointed to by *object*, and stores the subtraction result in the atomic object pointed to by *object*.

Return value

Returns atomically the value pointed to by *object* before the subtraction.

atomic_fetch_sub_explicit (C11)
Purpose

Performs the subtraction operation on an atomic object.

Prototypes

C atomic_fetch_sub_explicit(volatile *A* **object*, *M operand*, memory_order *order*);

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
A pointer that points to the atomic object to modify.

M The type of a specified operand is determined as follows:
- If *A* is an atomic integer type, *M* is the corresponding non-atomic type of *A*.
- If *A* is an atomic pointer type, *M* is the `ptrdiff_t` type.

operand
A specified value to subtract from the value in the atomic object.

order
The memory synchronization order for this operation.

Usage

This function subtracts the specified *operand* from the value pointed to by *object*, and stores the subtraction result in the atomic object pointed to by *object*.

Memory is affected according to the value of *order*.

Return value

Returns atomically the value pointed to by *object* before the subtraction.

atomic_fetch_or (C11)
Purpose

Performs the bitwise inclusive OR operation on an atomic object.

Prototypes

C atomic_fetch_or(volatile *A* **object*, *M operand*);

Parameters

A The type of an atomic object.

C The corresponding non-atomic type of *A*.

object
A pointer that points to the atomic object to modify.

M The type of a specified operand is determined as follows:
- If *A* is an atomic integer type, *M* is the corresponding non-atomic type of *A*.
- If *A* is an atomic pointer type, *M* is the `ptrdiff_t` type.

operand
The value to perform the bitwise inclusive OR operation on the value in the atomic object.

Usage

This function performs the bitwise inclusive OR operation on the value pointed to by *object* and the specified *operand*, and stores the result in the atomic object pointed to by *object*.

Return value

Returns atomically the value pointed to by *object* before the bitwise inclusive OR operation.

atomic_fetch_or_explicit (C11)
Purpose

Performs the bitwise inclusive OR operation on an atomic object.

Prototypes

C atomic_fetch_or_explicit(volatile *A* **object*, *M operand*, memory_order *order*);

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
A pointer that points to the atomic object to modify.

M The type of a specified operand is determined as follows:
- If *A* is an atomic integer type, *M* is the corresponding non-atomic type of *A*.
- If *A* is an atomic pointer type, *M* is the `ptrdiff_t` type.

operand
The value to perform the bitwise inclusive OR operation on the value in the atomic object.

order
The memory synchronization order for this operation.

Usage

This function performs the bitwise inclusive OR operation on the value pointed to by *object* and the specified *operand*, and stores the result in the atomic object pointed to by *object*.

Memory is affected according to the value of *order*.

Return value

Returns atomically the value pointed to by *object* before the bitwise inclusive OR operation.

atomic_fetch_xor (C11)
Purpose

Performs the bitwise exclusive OR operation on an atomic object.

Prototypes

C atomic_fetch_xor(volatile *A* **object*, *M operand*);

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
A pointer that points to the atomic object to modify.

M The type of a specified operand is determined as follows:
- If *A* is an atomic integer type, *M* is the corresponding non-atomic type of *A*.
- If *A* is an atomic pointer type, *M* is the `ptrdiff_t` type.

operand
A specified value to perform the bitwise exclusive OR operation on the value in the atomic object.

Usage

This function performs the bitwise exclusive OR operation on the value pointed to by *object* and the specified *operand*, and stores the result in the atomic object pointed to by *object*.

Return value

Returns atomically the value pointed to by *object* before the bitwise exclusive OR operation.

atomic_fetch_xor_explicit (C11)
Purpose

Performs the bitwise exclusive OR operation on an atomic object.

Prototypes

C atomic_fetch_xor_explicit(volatile *A* **object*, *M operand*, memory_order *order*);

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
A pointer that points to the atomic object to modify.

M The type of a specified operand is determined as follows:
- If *A* is an atomic integer type, *M* is the corresponding non-atomic type of *A*.
- If *A* is an atomic pointer type, *M* is the `ptrdiff_t` type.

operand
A specified value to perform the bitwise exclusive OR operation on the value in the atomic object.

order
The memory synchronization order for this operation.

Usage

This function performs the bitwise exclusive OR operation on the value pointed to by *object* and the specified *operand*, and stores the result in the atomic object pointed to by *object*.

Memory is affected according to the value of *order*. These operations are atomic read-modify-write operations.

Return value

Returns atomically the value pointed to by *object* before the bitwise exclusive OR operation.

atomic_fetch_and (C11)
Purpose

Performs the bitwise AND operation on an atomic object.

Prototypes

C atomic_fetch_and(volatile *A* **object*, *M operand*);

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
A pointer that points to the atomic object to modify.

M The type of a specified operand is determined as follows:

- If *A* is an atomic integer type, *M* is the corresponding non-atomic type of *A*.
- If *A* is an atomic pointer type, *M* is the `ptrdiff_t` type.

operand
A specified value to perform the bitwise AND operation on the value in the atomic object.

Usage

This function performs the bitwise AND operation on the value pointed to by *object* and the specified *operand*, and stores the result in the atomic object pointed to by *object*.

Return value

Returns atomically the value pointed to by *object* before the bitwise AND operation.

atomic_fetch_and_explicit (C11)
Purpose

Performs the bitwise AND operation on an atomic object.

Prototypes

C atomic_fetch_and_explicit(volatile *A* **object*, *M operand*, memory_order *order*);

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
A pointer that points to the atomic object to modify.

M The type of a specified operand is determined as follows:

- If *A* is an atomic integer type, *M* is the corresponding non-atomic type of *A*.
- If *A* is an atomic pointer type, *M* is the `ptrdiff_t` type.

operand
A specified value to perform the bitwise AND operation on the value in the atomic object.

order
The memory synchronization order for this operation.

Usage

This function performs the bitwise AND operation on the value pointed to by *object* and the specified *operand*, and stores the result in the atomic object pointed to by *object*.

Memory is affected according to the value of *order*.

Return value

Returns atomically the value pointed to by *object* before the bitwise AND operation.

atomic_flag_test_and_set (C11)
Purpose

Sets an `atomic_flag` object to true.

Prototypes

_Bool atomic_flag_test_and_set(volatile atomic_flag *_object_);

Parameter

`object`
The pointer that points to the `atomic_flag` object to modify.

Usage

This function atomically sets the value pointed to by *object* to true.

Return value

Returns atomically the value of the *object* immediately before the update.

atomic_flag_test_and_set_explicit (C11)
Purpose

Sets an `atomic_flag` object to true.

Prototypes

_Bool atomic_flag_test_and_set_explicit(volatile atomic_flag *_object_, memory_order _order_);

Parameters

`object`
The pointer that points to the atomic flag object to modify.

`order`
The memory synchronization order for this operation.

Usage

This function atomically sets the value pointed to by *object* to true.

Memory is affected according to the value of *order*.

Return value

Returns atomically the value of the *object* immediately before the update.

atomic_flag_clear (C11)
Purpose

Sets an `atomic_flag` object to false.

Prototypes

void atomic_flag_clear(volatile atomic_flag *_object_);

Parameter

`object`
The pointer that points to the `atomic_flag` object to modify.

Usage

This function atomically sets the value pointed to by *object* to false.

Return value

Returns no value.

atomic_flag_clear_explicit (C11)
Purpose

Sets an `atomic_flag` object to false.

Prototypes

void atomic_flag_clear_explicit(volatile atomic_flag *_object_, memory_order *order*);

Parameters

`object`
The pointer that points to the atomic flag object to modify.

`order`
The memory synchronization order for this operation. The *order* parameter cannot be `memory_order_acquire` or `memory_order_acq_rel`.

Usage

This function atomically sets the value pointed to by *object* to false.

Memory is affected according to the value of *order*.

Return value

Returns no value.

Atomic library (C++11)

The header <atomic> defines the following types, macros, and functions performing atomic operations on data that is shared among threads:

- "The memory_order type (C++11)"
- "Lock-free property (C++11)" on page 280
- "The atomic types (C++11)" on page 280
- "Operations on atomic types (C++11)" on page 296
- "Flag type and operations (C++11)" on page 308
- "Initialization (C++11)" on page 311
- "Memory synchronization order (C++11)" on page 312

Note: The keyword noexcept is currently not supported by the compiler, and is ignored if you use it.

The memory_order type (C++11)

The memory_order type

The enumerated type memory_order determines how nonatomic memory accesses are to be ordered around an atomic operation.

```
namespace std {
  typedef enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
  } memory_order;
}
```

Its enumeration constants define memory order constraints as follows:

memory_order_relaxed
There are no constraints on reordering of memory accesses around the atomic variable.

memory_order_consume
In the current thread, no reads that are dependent on the currently loaded value can be reordered before this load.

memory_order_acquire
In the current thread, no reads can be reordered before this load.

memory_order_release
In the current thread, no writes can be reordered after this store.

memory_order_acq_rel
In the current thread, no reads can be reordered before this load, and that no writes can be reordered after this store.

memory_order_seq_cst
The operation has the same semantics as the acquire-release operation. In addition, it has a sequentially consistent operation order.

Lock-free property (C++11)

The following atomic lock-free macros indicate the lock-free property of the corresponding atomic types (both signed and unsigned).

- `#define ATOMIC_BOOL_LOCK_FREE` *value*
- `#define ATOMIC_CHAR_LOCK_FREE` *value*
- `#define ATOMIC_CHAR16_T_LOCK_FREE` *value*
- `#define ATOMIC_CHAR32_T_LOCK_FREE` *value*
- `#define ATOMIC_WCHAR_T_LOCK_FREE` *value*
- `#define ATOMIC_SHORT_LOCK_FREE` *value*
- `#define ATOMIC_INT_LOCK_FREE` *value*
- `#define ATOMIC_LONG_LOCK_FREE` *value*
- `#define ATOMIC_LLONG_LOCK_FREE` *value*
- `#define ATOMIC_POINTER_LOCK_FREE` *value*

The values of the atomic lock-free macros are defined as follows:

0 Indicates that the atomic type is never lock-free.

1 Indicates that the atomic type is sometimes lock-free.

2 Indicates that the atomic type is always lock-free.

The atomic types (C++11)

Objects of the atomic types are free from data races. You can create atomic objects by using the following atomic types:

```
template< class T > struct atomic;      //The generic atomic type


template < > struct atomic<Integral>;   //Full specializations for integral types


template< class T > struct atomic<T*>;  //Partial specializations for all pointer
                                        //types
```

Specializations and instantiations of the atomic template must have a deleted copy constructor, a deleted copy assignment operator, and a `constexpr` value constructor.

The standard library provides full specializations of the `atomic` template for integral types, one full specialization for the `bool` type, and partial specializations for all pointer types. Each of these specializations has standard layout, a trivial default constructor, and a trivial destructor. They all support aggregate initialization syntax.

The following table shows typedef names and the corresponding full specializations for the `bool` type and integral types.

Table 34. Typedef names and corresponding full specializations

Typedef name	Full specialization
`atomic_bool`	`atomic<bool>`
`atomic_char`	`atomic<char>`
`atomic_schar`	`atomic<schar>`
`atomic_uchar`	`atomic<unsigned char>`
`atomic_short`	`atomic<short>`

Table 34. Typedef names and corresponding full specializations (continued)

Typedef name	Full specialization
atomic_ushort	atomic<unsigned short>
atomic_int	atomic<int>
atomic_uint	atomic<unsigned int>
atomic_long	atomic<long>
atomic_ulong	atomic<unsigned long>
atomic_llong	atomic<long long>
atomic_ullong	atomic<unsigned long long>
atomic_char16_t	atomic<char16_t>
atomic_char32_t	atomic<char32_t>
atomic_wchar_t	atomic<wchar_t>
atomic_int_least8_t	atomic<int_least8_t>
atomic_uint_least8_t	atomic<uint_least8_t>
atomic_int_least16_t	atomic<int_least16_t>
atomic_uint_least16_t	atomic<uint_least16_t>
atomic_int_least32_t	atomic<int_least32_t>
atomic_uint_least32_t	atomic<uint_least32_t>
atomic_int_least64_t	atomic<int_least64_t>
atomic_uint_least64_t	atomic<uint_least64_t>
atomic_int_fast8_t	atomic<int_fast8_t>
atomic_uint_fast8_t	atomic<uint_fast8_t>
atomic_int_fast16_t	atomic<int_fast16_t>
atomic_uint_fast16_t	atomic<uint_fast16_t>
atomic_int_fast32_t	atomic<int_fast32_t>
atomic_uint_fast32_t	atomic<uint_fast32_t>
atomic_int_fast64_t	atomic<int_fast64_t>
atomic_uint_fast64_t	atomic<uint_fast64_t>
atomic_intptr_t	atomic<intptr_t>
atomic_uintptr_t	atomic<uintptr_t>
atomic_size_t	atomic<size_t>
atomic_ptrdiff_t	atomic<ptrdiff_t>
atomic_intmax_t	atomic<intmax_t>
atomic_uintmax_t	atomic<uintmax_t>

Synopsis

```
namespace std {
  template <class T> struct atomic {
    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;

    void store(T, memory_order = memory_order_seq_cst) volatile noexcept;
    void store(T, memory_order = memory_order_seq_cst) noexcept;

    T load(memory_order = memory_order_seq_cst) const volatile noexcept;
    T load(memory_order = memory_order_seq_cst) const noexcept;
```

```
    operator T() const volatile noexcept;
    operator T() const noexcept;

    T exchange(T, memory_order = memory_order_seq_cst) volatile noexcept;
    T exchange(T, memory_order = memory_order_seq_cst) noexcept;

    bool compare_exchange_weak(T&, T, memory_order, memory_order) volatile
    noexcept;
    bool compare_exchange_weak(T&, T, memory_order, memory_order) noexcept;
    bool compare_exchange_strong(T&, T, memory_order, memory_order) volatile
    noexcept;
    bool compare_exchange_strong(T&, T, memory_order, memory_order) noexcept;
    bool compare_exchange_weak(T&, T, memory_order = memory_order_seq_cst)
    volatile noexcept;
    bool compare_exchange_weak(T&, T, memory_order = memory_order_seq_cst)
    noexcept;
    bool compare_exchange_strong(T&, T, memory_order = memory_order_seq_cst)
    volatile noexcept;
    bool compare_exchange_strong(T&, T, memory_order = memory_order_seq_cst)
    noexcept;

    atomic() noexcept = default;
    constexpr atomic(T) noexcept;
    atomic(const atomic&) = delete;

    atomic& operator=(const atomic&) = delete;
    atomic& operator=(const atomic&) volatile = delete;

    T operator=(T) volatile noexcept;
    T operator=(T) noexcept;
};

  template <> struct atomic<integral > {
    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;

    void store(integral , memory_order = memory_order_seq_cst)
    volatile noexcept;
    void store(integral , memory_order = memory_order_seq_cst) noexcept;

    integral load(memory_order = memory_order_seq_cst) const volatile noexcept;
    integral load(memory_order = memory_order_seq_cst) const noexcept;

    operator integral() const volatile noexcept;
    operator integral() const noexcept;

    integral exchange(integral , memory_order = memory_order_seq_cst)
    volatile noexcept;
    integral exchange(integral , memory_order = memory_order_seq_cst)
    noexcept;

    bool compare_exchange_weak(integral &, integral , memory_order,
    memory_order) volatile noexcept;
    bool compare_exchange_weak(integral &, integral , memory_order,
    memory_order) noexcept;
    bool compare_exchange_strong(integral &, integral , memory_order,
    memory_order) volatile noexcept;
    bool compare_exchange_strong(integral &, integral , memory_order,
    memory_order) noexcept;
    bool compare_exchange_weak(integral &, integral , memory_order =
    memory_order_seq_cst) volatile noexcept;
    bool compare_exchange_weak(integral &, integral , memory_order =
    memory_order_seq_cst) noexcept;
    bool compare_exchange_strong(integral &, integral , memory_order =
    memory_order_seq_cst) volatile noexcept;
    bool compare_exchange_strong(integral &, integral , memory_order =
```

```
    memory_order_seq_cst) noexcept;

    integral fetch_add(integral , memory_order = memory_order_seq_cst)
    volatile noexcept;
    integral fetch_add(integral , memory_order = memory_order_seq_cst)
    noexcept;
    integral fetch_sub(integral , memory_order = memory_order_seq_cst)
    volatile noexcept;
    integral fetch_sub(integral , memory_order = memory_order_seq_cst)
    noexcept;
    integral fetch_and(integral , memory_order = memory_order_seq_cst)
    volatile noexcept;
    integral fetch_and(integral , memory_order = memory_order_seq_cst)
    noexcept;
    integral fetch_or(integral , memory_order = memory_order_seq_cst)
    volatile noexcept;
    integral fetch_or(integral , memory_order = memory_order_seq_cst)
    noexcept;
    integral fetch_xor(integral , memory_order = memory_order_seq_cst)
    volatile noexcept;
    integral fetch_xor(integral , memory_order = memory_order_seq_cst)
    noexcept;

    atomic() noexcept = default;
    constexpr atomic(integral ) noexcept;
    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    atomic& operator=(const atomic&) volatile = delete;

    integral operator=(integral ) volatile noexcept;
    integral operator=(integral ) noexcept;
    integral operator++(int) volatile noexcept;
    integral operator++(int) noexcept;
    integral operator--(int) volatile noexcept;
    integral operator--(int) noexcept;
    integral operator++() volatile noexcept;
    integral operator++() noexcept;
    integral operator--() volatile noexcept;
    integral operator--() noexcept;
    integral operator+=(integral ) volatile noexcept;
    integral operator+=(integral ) noexcept;
    integral operator-=(integral ) volatile noexcept;
    integral operator-=(integral ) noexcept;
    integral operator&=(integral ) volatile noexcept;
    integral operator&=(integral ) noexcept;
    integral operator|=(integral ) volatile noexcept;
    integral operator|=(integral ) noexcept;
    integral operator^=(integral ) volatile noexcept;
    integral operator^=(integral ) noexcept;
};


  template <class T> struct atomic<T*> {
    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;

    void store(T*, memory_order = memory_order_seq_cst) volatile noexcept;
    void store(T*, memory_order = memory_order_seq_cst) noexcept;

    T* load(memory_order = memory_order_seq_cst) const volatile noexcept;
    T* load(memory_order = memory_order_seq_cst) const noexcept;

    operator T*() const volatile noexcept;
    operator T*() const noexcept;

    T* exchange(T*, memory_order = memory_order_seq_cst) volatile noexcept;
    T* exchange(T*, memory_order = memory_order_seq_cst) noexcept;
```

```
    bool compare_exchange_weak(T*&, T*, memory_order, memory_order) volatile
    noexcept;
    bool compare_exchange_weak(T*&, T*, memory_order, memory_order) noexcept;
    bool compare_exchange_strong(T*&, T*, memory_order, memory_order) volatile
    noexcept;
    bool compare_exchange_strong(T*&, T*, memory_order, memory_order) noexcept;
    bool compare_exchange_weak(T*&, T*, memory_order = memory_order_seq_cst)
    volatile noexcept;
    bool compare_exchange_weak(T*&, T*, memory_order = memory_order_seq_cst)
    noexcept;
    bool compare_exchange_strong(T*&, T*, memory_order = memory_order_seq_cst)
    volatile noexcept;
    bool compare_exchange_strong(T*&, T*, memory_order = memory_order_seq_cst)
    noexcept;

    T* fetch_add(ptrdiff_t, memory_order = memory_order_seq_cst)
    volatile noexcept;
    T* fetch_add(ptrdiff_t, memory_order = memory_order_seq_cst)
    noexcept;
    T* fetch_sub(ptrdiff_t, memory_order = memory_order_seq_cst)
    volatile noexcept;
    T* fetch_sub(ptrdiff_t, memory_order = memory_order_seq_cst)
    noexcept;

    atomic() noexcept = default;
    constexpr atomic(T*) noexcept;
    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    atomic& operator=(const atomic&) volatile = delete;

    T* operator=(T*) volatile noexcept;
    T* operator=(T*) noexcept;
    T* operator++(int) volatile noexcept;
    T* operator++(int) noexcept;
    T* operator--(int) volatile noexcept;
    T* operator--(int) noexcept;
    T* operator++() volatile noexcept;
    T* operator++() noexcept;
    T* operator--() volatile noexcept;
    T* operator--() noexcept;
    T* operator+=(ptrdiff_t) volatile noexcept;
    T* operator+=(ptrdiff_t) noexcept;
    T* operator-=(ptrdiff_t) volatile noexcept;
    T* operator-=(ptrdiff_t) noexcept;
};
}
```

Member functions (C++11)

Constructors (C++11):

atomic::atomic (C++11):
Purpose

Constructs an atomic object.

Prototypes

atomic() = default;

constexpr atomic(*T desired*) noexcept;

atomic(const atomic&) = delete;

Parameter

desired
The value to initialize the atomic object

Usage

The first constructor leaves the atomic object in an uninitialized state.

The second constructor initializes the object with the value *desired*. The initialization is not an atomic operation.

The third constructor specifies that the atomic variables cannot be copy-constructed (copied).

atomic::operator= (C++11):
Purpose

Uses a specified value *desired* to replace the stored value.

Prototypes

C A ::operator=(*C desired*) volatile noexcept;

C A ::operator=(*C desired*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

desired
The value to be assigned.

Return value

Returns the value *desired*.

atomic::is_lock_free (C++11):
Purpose

Checks whether the atomic operations on the object are lock-free.

Prototypes

bool *A* ::is_lock_free() const volatile noexcept;

bool *A* ::is_lock_free() const noexcept;

Parameter

A One of the atomic types.

Return value

Returns true if the operations on the object are lock-free. Otherwise, false is returned.

atomic::store (C++11):
Purpose

Uses a specified value to replace the stored value.

Memory is affected according to the value of *order*.

Prototypes

void *A* ::store(*C desired*, memory_order *order* = memory_order_seq_cst) volatile noexcept;

void *A* ::store(*C desired*, memory_order *order* = memory_order_seq_cst) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

desired
The value to store into the atomic variable.

order
The memory order constraints to enforce. The *order* parameter cannot be `memory_order_relaxed`, `memory_order_release`, or `memory_order_seq_cst`.

atomic::load (C++11):
Purpose

Obtains the value of an atomic object.

Memory is affected according to the value of *order*.

Prototypes

C A ::load(memory_order *order* = memory_order_seq_cst) const volatile noexcept;

C A ::load(memory_order *order* = memory_order_seq_cst) const noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

order
The memory order constraints to enforce. The *order* parameter cannot be `memory_order_release` or `memory_order_acq_rel`.

Return value

Returns the value that is stored in **this*.

atomic::operator C (C++11):
Purpose

Atomically loads and returns the current value of the atomic variable.

This function is equivalent to `load()`.

Prototypes

A ::operator *C*() const volatile noexcept;

A ::operator *C*() const noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

Return value

Returns the value that is stored in **this*.

atomic::exchange (C++11):
Purpose

Atomically replaces the value of the atomic object.

Memory is affected according to the value of *order*.

Prototypes

C A ::exchange(*C desired*, memory_order *order* = memory_order_seq_cst) volatile noexcept;

C A ::exchange(*C desired*, memory_order *order* = memory_order_seq_cst) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

desired
The value to be assigned.

order
The memory order constraints to enforce.

Return value

Returns the value of the atomic variable immediately before the exchange.

atomic::compare_exchange_weak (C++11):
Purpose

Atomically compares the value pointed to by *this* with the value in *expected*, and performs the following operations based on the comparison results:

- If the comparison result is true, the function replaces the value pointed to by *this* with the value *desired*.
- If the comparison result is false, the function updates the value in *expected* with the value **this*.

Prototypes

bool *A* ::compare_exchange_weak(*C* & *expected*, *C desired*, memory_order *success*, memory_order *failure*) volatile noexcept;

bool *A* ::compare_exchange_weak(*C* & *expected*, *C desired*, memory_order *success*, memory_order *failure*) noexcept;

bool *A* ::compare_exchange_weak(*C* & *expected*, *C desired*, memory_order *order* = memory_order_seq_cst) volatile noexcept;

bool *A* ::compare_exchange_weak(*C* & *expected, C desired*, memory_order *order* = memory_order_seq_cst) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

expected
A reference that points to the value expected to be found in the atomic object.

desired
The value to store in the atomic object if it is as expected.

success
The memory synchronization order for the read-modify-write operation if the comparison result is true.

failure
The memory synchronization order for the load operation if the comparison result is false. The `failure` parameter cannot be `memory_order_release` or `memory_order_acq_rel`. The *failure* parameter cannot be stronger than the *success* parameter.

order
The memory synchronization order for both operations.

Return value

Returns the comparison result.

atomic::compare_exchange_strong (C++11):
Purpose

Atomically compares the value pointed to by *this* with the value pointed to by *expected*, and performs the following operations based on the comparison results:

- If the comparison result is true, the function replaces the value pointed to by *this* with the value *desired*.
- If the comparison result is false, the function updates the value in *expected* with the value **this* .

Prototypes

bool A ::compare_exchange_strong(*C* & *expected*, *C desired*, memory_order *success*, memory_order *failure*) volatile noexcept;

bool A ::compare_exchange_strong(*C* & *expected*, *C desired*, memory_order *success*, memory_order *failure*) noexcept;

bool A ::compare_exchange_strong(*C* & *expected*, *C desired*, memory_order *order* = memory_order_seq_cst) volatile noexcept;

bool A ::compare_exchange_strong(*C* & *expected*, *C desired*, memory_order *order* = memory_order_seq_cst) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

expected
A reference that points to the value expected to be found in the atomic object.

desired
The value to store in the atomic object if it is as expected.

success
The memory synchronization order for the read-modify-write operation if the comparison result is true.

failure
The memory synchronization order for the load operation if the comparison result is false. The `failure` parameter cannot be `memory_order_release` or `memory_order_acq_rel`. The *failure* parameter cannot be stronger than the *success* parameter.

order
The memory synchronization order for both operations.

Return value

Returns the comparison result.

Specialized member functions (C++11)

atomic::fetch_add (C++11):
Purpose

Atomically adds a value to an existing value that is stored in an atomic object.

Memory is affected according to the value of *order*.

Prototypes

C A ::fetch_add (*M operand*, memory_order *order* = memory_order_seq_cst) volatile noexcept;

C A ::fetch_add (*M operand*, memory_order *order* = memory_order_seq_cst) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

M *M* is the type of the other argument for the addition.
- For integral atomic types, *M* is *C*.
- For atomic address types, *M* is `std::ptrdiff_t`.

operand
The other argument of the addition.

order
The memory order constraints to enforce.

Return value

Returns the value of the atomic variable before the addition.

atomic::fetch_sub (C++11):
Purpose

Subtracts a specified value from the stored value.

Memory is affected according to the value of *order*.

Prototypes

C A ::fetch_sub (*M operand*, memory_order *order* = memory_order_seq_cst) volatile noexcept;

C A ::fetch_sub (*M operand*, memory_order *order* = memory_order_seq_cst) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

M M is the type of the other argument for the subtraction.
- For integral atomic types, *M* is *C*.
- For atomic address types, *M* is `std::ptrdiff_t`.

operand
The other argument of the subtraction.

order
The memory order constraints to enforce.

Return value

Returns the value of the atomic variable before the subtraction.

atomic::fetch_and (C++11):
Purpose

Performs a bitwise AND operation on a value and an existing value that is stored in an atomic object.

Memory is affected according to the value of *order*.

Prototypes

C A ::fetch_and (*M operand*, memory_order *order* = memory_order_seq_cst) volatile noexcept;

C A ::fetch_and (*M operand*, memory_order *order* = memory_order_seq_cst) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

M M is the type of the other argument for the bitwise AND operation.

- For integral atomic types, *M* is *C*.
- For atomic address types, *M* is `std::ptrdiff_t`.

operand
The other argument of the bitwise AND operation.

order
The memory order constraints to enforce.

Return value

Returns the value of the atomic variable before the bitwise AND operation.

atomic::fetch_or (C++11):
Purpose

Performs a bitwise inclusive OR operation on a value and an existing value that is stored in an atomic object.

Memory is affected according to the value of *order*.

Prototypes

C A ::fetch_or (*M operand*, memory_order *order* = memory_order_seq_cst) volatile noexcept;

C A ::fetch_or (*M operand*, memory_order *order* = memory_order_seq_cst) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

M *M* is the type of the other argument for the bitwise inclusive OR operation.

- For integral atomic types, *M* is *C*.
- For atomic address types, *M* is `std::ptrdiff_t`.

operand
The other argument of the bitwise inclusive OR operation.

order
The memory order constraints to enforce.

Return value

Returns the value of the atomic variable before the bitwise inclusive OR operation.

atomic::fetch_xor (C++11):
Purpose

Performs a bitwise exclusive OR operation on a value and an existing value that is stored in an atomic object.

Memory is affected according to the value of *order*.

Prototypes

C A ::fetch_xor (*M operand*, memory_order *order* = memory_order_seq_cst) volatile noexcept;

C A ::fetch_xor (*M operand*, memory_order *order* = memory_order_seq_cst) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

M *M* is the type of the other argument for the bitwise exclusive OR operation.
- For integral atomic types, *M* is *C*.
- For atomic address types, *M* is `std::ptrdiff_t`.

operand
The other argument of the bitwise exclusive OR operation.

order
The memory order constraints to enforce.

Return value

Returns the value of the atomic variable before the bitwise exclusive OR operation.

atomic::operator++ (C++11):
Purpose

Increments the atomic value by one.

Prototypes

C A ::operator++() volatile noexcept;

C A ::operator++() noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

Return value

Returns the value of the atomic variable after the modification.

atomic::operator++(int) (C++11):
Purpose

Increments the atomic value by one.

Prototypes

C A ::operator++(int) volatile noexcept;

C A ::operator++(int) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

Return value

Returns the value of the atomic variable before the modification.

atomic::operator-- (C++11):
Purpose

Decrements the atomic value by one.

Prototypes

C A ::operator--() volatile noexcept;

C A ::operator--() noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

Return value

Returns the value of the atomic variable after the modification.

atomic::operator--(int) (C++11):
Purpose

Decrements the atomic value by one.

Prototypes

C A ::operator--(int) volatile noexcept;

C A ::operator--(int) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

Return value

Returns the value of the atomic variable before the modification.

atomic::operator+= (C++11):
Purpose

Adds a specified value to the stored value.

Prototypes

C A ::operator + =(*M operand*) volatile noexcept;

C A ::operator + =(*M operand*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

M *M* is the type of the other argument for the addition.
- For integral atomic types, *M* is *C*.
- For atomic address types, *M* is `std::ptrdiff_t`.

Operand
The specified argument for the addition.

Return value

Return the result of the addition.

atomic::operator-= (C++11):
Purpose

Subtracts a specified value from the stored value.

Prototypes

C A ::operator - =(*M operand*) volatile noexcept;

C A ::operator - =(*M operand*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

M *M* is the type of the other argument for the subtraction.
- For integral atomic types, *M* is *C*.
- For atomic address types, *M* is `std::ptrdiff_t`.

operand
The specified argument for the subtraction.

Return value

Returns the result of the subtraction.

atomic::operator&= (C++11):
Purpose

Performs a bitwise AND operation on a specified value and the value **this*.

Prototypes

C A ::operator & =(*M operand*) volatile noexcept;

C A ::operator & =(*M operand*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

M *M* is the type of the other argument for the bitwise AND operation.
- For integral atomic types, *M* is *C*.
- For atomic address types, *M* is `std::ptrdiff_t`.

operand
The specified argument for the bitwise AND operation.

Return value

Returns the result of the bitwise AND operation.

atomic::operator| = (C++11):
Purpose

Performs a bitwise OR operation on a specified value and the value stored in **this*.

Prototypes

C A ::operator | =(*M operand*) volatile noexcept;

C A ::operator | =(*M operand*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

M *M* is the type of the other argument for the bitwise OR operation.
- For integral atomic types, *M* is *C*.
- For atomic address types, *M* is `std::ptrdiff_t`.

operand
The specified argument for the bitwise OR operation.

Return value

Returns the result of the bitwise OR operation.

atomic::operator^ = (C++11):
Purpose

Performs a bitwise exclusive OR operation on a specified value and the value stored in **this*.

Prototypes

C A ::operator ^ =(*M operand*) volatile noexcept;

C A ::operator ^ =(*M operand*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

M *M* is the type of the other argument for the bitwise exclusive OR operation.
- For integral atomic types, *M* is *C*.
- For atomic address types, *M* is `std::ptrdiff_t`.

operand
The specified argument for the bitwise exclusive OR operation.

Return value

Returns the result of the bitwise exclusive OR operation.

Operations on atomic types (C++11)

atomic_is_lock_free (C++11)
Purpose

Checks whether the atomic operations on the object are lock-free.

Prototypes

bool atomic_is_lock_free(const volatile *A *object*) noexcept;

bool atomic_is_lock_free(const *A *object*) noexcept;

Parameter

A One of the atomic types.

object
The pointer that points to the atomic object to inspect.

Return value

Returns true if the operations on the object are lock-free. Otherwise, false is returned.

atomic_store (C++11)
Purpose

Uses a specified value to replace the stored value.

Prototypes

void atomic_store(volatile *A* * *object*, *C desired*) noexcept;

void atomic_store(*A* * *object*, *C desired*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
The pointer that points to the atomic object to modify.

desired
The value to store into the atomic object.

atomic_store_explicit (C++11)
Purpose

Uses a specified value to replace the stored value.

Memory is affected according to the value of *order*.

Prototypes

void atomic_store_explicit(volatile *A* **object*, *C desired*, memory_order *order*) noexcept;

void atomic_store_explicit(*A* * *object*, *C desired*, memory_order *order*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
The pointer that points to the atomic object to modify.

desired
The value to store into the atomic object.

order
Memory order constraints to enforce. The *order* parameter cannot be `memory_order_relaxed`, `memory_order_release`, or `memory_order_seq_cst`.

atomic_load (C++11)
Purpose

Atomically obtains the value of the atomic object.

Prototypes

C atomic_load(const volatile *A* * *object*) noexcept;

C atomic_load(const *A* * *object*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
A pointer that points to the atomic object to access.

Return value

Returns the value that is pointed to by *object*.

atomic_load_explicit (C++11)
Purpose

Atomically obtains the value of the atomic object.

Memory is affected according to the value of *order*.

Prototypes

C atomic_load_explicit(const volatile *A* * *object*, memory_order *order*) noexcept;

C atomic_load_explicit(const *A* * *object*, memory_order *order*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
A pointer that points to the atomic object to access.

order
Memory order constraints to enforce. The *order* argument cannot be `memory_order_release` or `memory_order_acq_rel`.

Return value

Returns the value that is pointed to by *object*.

atomic_exchange (C++11)
Purpose

Atomically replaces the value of the atomic object and obtains the value held previously.

Prototypes

C atomic_exchange(volatile *A* * *object*, *C desired*) noexcept;

C atomic_exchange(*A* * *object*, C *desired*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
A pointer that points to the atomic object to modify.

desired
The value to be assigned.

Return value

Returns the value pointed to by `object` immediately before the exchange.

atomic_exchange_explicit (C++11)

Purpose

Atomically replaces the value of the atomic object and obtains the value held previously.

Memory is affected according to the value of *order*.

Prototypes

C atomic_exchange_explicit(volatile *A* * *object*, *C desired*, memory_order *order*) noexcept;

C atomic_exchange_explicit(*A* * *object*, *C desired*, memory_order *order*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
A pointer that points to the atomic object to modify.

desired
The value to be assigned.

order
Memory order constraints to enforce.

Return value

Returns the value that is pointed to by `object` immediately before the exchange.

atomic_compare_exchange_weak (C++11)

Purpose

Atomically compares the value pointed to by *object* with the value pointed to by *expected*, and performs the following operations based on the comparison results:

- Replaces the value pointed to by *object* with the value *desired* if the comparison result is true.
- Updates the value pointed to by *expected* with the value pointed to by *object* if the comparison result is false.

Prototypes

bool atomic_compare_exchange_weak(volatile *A* * *object*, *C* * *expected*, *C desired*) noexcept;

bool atomic_compare_exchange_weak(*A* * *object*, *C* * *expected*, *C desired*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
The pointer that points to the atomic object to test and modify.

expected
The pointer that points to the value expected to be found in the atomic object.

desired
The value to store in the atomic object if it is as expected.

Return value

Returns the result of the comparison.

atomic_compare_exchange_weak_explicit (C++11)

Purpose

Atomically compares the value pointed to by *object* with the value pointed to by *expected*, and performs the following operations based on the comparison results:

- Replaces the value pointed to by *object* with the value *desired* if the comparison result is true.
- Updates the value pointed to by *expected* with the value pointed to by *object* if the comparison result is false.

Prototypes

bool atomic_compare_exchange_weak_explicit(volatile *A* * *object*, C * *expected*, *C desired*, memory_order *success*, memory_order *failure*) noexcept;

bool atomic_compare_exchange_weak_explicit(*A* * *object*, *C* * *expected*, *C desired*, memory_order *success*, memory_order *failure*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
The pointer that points to the atomic object to test and modify.

expected
The pointer that points to the value expected to be found in the atomic object.

desired
The value to store in the atomic object if it is as expected.

success
The memory synchronization order for the read-modify-write operation if the comparison result is true.

failure
The memory synchronization order for the load operation if the comparison result is false. The `failure` parameter cannot be `memory_order_release` or `memory_order_acq_rel`. The *failure* parameter cannot be stronger than the *success* parameter.

Return value

Returns the result of the comparison.

atomic_compare_exchange_strong (C++11)
Purpose

Atomically compares the value pointed to by *object* with the value pointed to by *expected*, and performs the following operations based on the comparison results:

- Replaces the value pointed to by *object* with the value *desired* if the comparison result is true.
- Updates the value pointed to by *expected* with the value pointed to by *object* if the comparison result is false.

Prototypes

bool atomic_compare_exchange_strong(volatile *A * object*, *C * expected*, *C desired*) noexcept;

bool atomic_compare_exchange_strong(*A * object*, *C * expected*, *C desired*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
The pointer that points to the atomic object to test and modify.

expected
The pointer that points to the value expected to be found in the atomic object.

desired
The value to store in the atomic object if it is as expected.

Return value

Returns the result of the comparison.

atomic_compare_exchange_strong_explicit (C++11)
Purpose

Atomically compares the value pointed to by *object* with the value pointed to by *expected*, and performs the following operations based on the comparison results:

- Replaces the value pointed to by *object* with the value *desired* if the comparison result is true.

- Updates the value pointed to by *expected* with the value pointed to by *object* if the comparison result is false.

Prototypes

bool atomic_compare_exchange_strong_explicit(volatile *A* * *object*, *C* * *expected*, *C desired*, memory_order *success*, memory_order *failure*) noexcept;

bool atomic_compare_exchange_strong_explicit(*A* * *object*, *C* * *expected*, *C desired*, memory_order *success*, memory_order *failure*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
The pointer that points to the atomic object to test and modify.

expected
The pointer that points to the value expected to be found in the atomic object.

desired
The value to store in the atomic object if it is as expected.

success
The memory synchronization order for the read-modify-write operation if the comparison result is true.

failure
The memory synchronization order for the load operation if the comparison result is false. The `failure` parameter cannot be `memory_order_release` or `memory_order_acq_rel`. The *failure* parameter cannot be stronger than the *success* parameter.

Return value

Returns the result of the comparison.

atomic_fetch_add (C++11)
Purpose

Atomically adds a value to the value pointed to by *object*.

Prototypes

C atomic_fetch_add (volatile *A* **object*, *M operand*) noexcept;

C atomic_fetch_add (*A* * *object*, *M operand*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

M *M* is the type of the other argument for the addition.
- For integral atomic types, *M* is *C*.
- For atomic address types, *M* is `std::ptrdiff_t`.

object
A pointer that points to the atomic object to modify.

operand
The other argument of the addition.

Return value

Returns the value of the atomic variable before the addition.

atomic_fetch_add_explicit (C++11)

Purpose

Atomically adds a value to the value pointed to by *object*.

Memory is affected according to the value of *order*.

Prototypes

C atomic_fetch_add_explicit(volatile *A* **object*, *M operand*, memory_order *order*) noexcept;

C atomic_fetch_add_explicit(*A* * *object*, *M operand*, memory_order *order*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

M *M* is the type of the other argument for the addition.
- For integral atomic types, *M* is *C*.
- For atomic address types, *M* is `std::ptrdiff_t`.

object
A pointer that points to the atomic object to modify.

operand
The other argument of the addition.

order
The memory order constraints to enforce.

Return value

Returns the value of the atomic variable before the addition.

atomic_fetch_sub (C++11)

Purpose

Subtracts a specified value from the value pointed to by *object*.

Prototypes

C atomic_fetch_sub (volatile *A* **object*, *M operand*) noexcept;

C atomic_fetch_sub (*A* * *object*, *M operand*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

M *M* is the type of the other argument for the subtraction.
- For integral atomic types, *M* is *C*.
- For atomic address types, *M* is `std::ptrdiff_t`.

object
A pointer that points to the atomic object to modify.

operand
The other argument of the subtraction.

Return value

Returns the value of the atomic variable before the subtraction.

atomic_fetch_sub_explicit (C++11)

Purpose

Subtracts a specified value from the value pointed to by *object*.

Memory is affected according to the value of *order*.

Prototypes

C atomic_fetch_sub_explicit(volatile *A* **object*, *M operand*, memory_order *order*) noexcept;

C atomic_fetch_sub_explicit(*A* * *object*, *M operand*, memory_order *order*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

M *M* is the type of the other argument for the subtraction.
- For integral atomic types, *M* is *C*.
- For atomic address types, *M* is `std::ptrdiff_t`.

object
A pointer that points to the atomic object to modify.

operand
The other argument of the subtraction.

order
The memory order constraints to enforce.

Return value

Returns the value of the atomic variable before the subtraction.

atomic_fetch_and (C++11)

Purpose

Performs a bitwise AND operation on a specified value and the value pointed to by *object*.

Prototypes

C atomic_fetch_and(volatile *A* **object*, *M operand*) noexcept;

C atomic_fetch_and(*A* * *object*, *M operand*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

M *M* is the type of the other argument for the bitwise AND operation.

- For integral atomic types, *M* is *C*.
- For atomic address types, *M* is `std::ptrdiff_t`.

object
A pointer that points to the atomic object to modify.

operand
The other argument of the bitwise AND operation.

Return value

Returns the value of the atomic variable before the bitwise AND operation.

atomic_fetch_and_explicit (C++11)

Purpose

Performs a bitwise AND operation on a specified value and the value pointed to by *object*.

Memory is affected according to the value of *order*.

Prototypes

C atomic_fetch_and_explicit(volatile *A* **object*, *M operand*, memory_order *order*) noexcept;

C atomic_fetch_and_explicit(*A* * *object*, *M operand*, memory_order *order*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

M *M* is the type of the other argument for the bitwise AND operation.

- For integral atomic types, *M* is *C*.
- For atomic address types, *M* is `std::ptrdiff_t`.

object
A pointer that points to the atomic object to modify.

operand
The other argument of the bitwise AND operation.

order
The memory order constraints to enforce.

Return value

Returns the value of the atomic variable before the bitwise AND operation.

atomic_fetch_or (C++11)
Purpose

Performs a bitwise inclusive OR operation on a specified value and the value pointed to by *object*.

Prototypes

C atomic_fetch_or(volatile *A* **object*, *M operand*) noexcept;

C atomic_fetch_or(*A * object*, *M operand*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

M *M* is the type of the other argument for the bitwise inclusive OR operation.
- For integral atomic types, *M* is *C*.
- For atomic address types, *M* is `std::ptrdiff_t`.

object
A pointer that points to the atomic object to modify.

operand
The other argument of the bitwise inclusive OR operation.

Return value

Returns the value of the atomic variable before the bitwise inclusive OR operation.

atomic_fetch_or_explicit (C++11)
Purpose

Performs a bitwise inclusive OR operation on a specified value and the value pointed to by *object*.

Memory is affected according to the value of *order*.

Prototypes

C atomic_fetch_or_explicit(volatile *A* **object*, *M operand*, memory_order *order*) noexcept;

C atomic_fetch_or_explicit(*A * object*, *M operand*, memory_order *order*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

M *M* is the type of the other argument for the bitwise inclusive OR operation.

- For integral atomic types, *M* is *C*.
- For atomic address types, *M* is `std::ptrdiff_t`.

object
A pointer that points to the atomic object to modify.

operand
The other argument of the bitwise inclusive OR operation.

order
The memory order constraints to enforce.

Return value

Returns the value of the atomic variable before the bitwise inclusive OR operation.

atomic_fetch_xor (C++11)
Purpose

Performs a bitwise exclusive OR operation on a specified value and the value pointed to by *object*.

Prototypes

C atomic_fetch_xor(volatile *A* **object*, *M operand*) noexcept;

C atomic_fetch_xor(*A * object*, *M operand*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

M *M* is the type of the other argument for the bitwise exclusive OR operation.

- For integral atomic types, *M* is *C*.
- For atomic address types, *M* is `std::ptrdiff_t`.

object
A pointer that points to the atomic object to modify.

operand
The other argument of the bitwise exclusive OR operation.

Return value

Returns the value of the atomic variable before the bitwise exclusive OR operation.

atomic_fetch_xor_explicit (C++11)
Purpose

Performs a bitwise exclusive OR operation on a specified value and the value **this*.

Memory is affected according to the value of *order*.

Prototypes

C atomic_fetch_xor_explicit(volatile *A* **object*, *M operand*, memory_order *order*) noexcept;

C atomic_fetch_xor_explicit(*A * object*, *M operand*, memory_order *order*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

M *M* is the type of the other argument for the bitwise exclusive OR operation.

- For integral atomic types, *M* is *C*.
- For atomic address types, *M* is `std::ptrdiff_t`.

object
A pointer that points to the atomic object to modify.

operand
The other argument of the bitwise exclusive OR operation.

order
The memory order constraints to enforce.

Return value

Returns the value of the atomic variable before the bitwise exclusive OR operation.

Flag type and operations (C++11)

The `atomic_flag` type (C++11)

`atomic_flag` is an atomic boolean type, and provides the classic test-and-set function. It has two states, set and clear.

Operations on an object of type `atomic_flag` must be lock-free.

Synopsis

```
typedef struct atomic_flag {
  bool test_and_set(memory_order = memory_order_seq_cst) volatile noexcept;
  bool test_and_set(memory_order = memory_order_seq_cst) noexcept;

  void clear(memory_order = memory_order_seq_cst) volatile noexcept;
  void clear(memory_order = memory_order_seq_cst) noexcept;

  atomic_flag() noexcept = default;
  atomic_flag(const atomic_flag&) = delete;

  atomic_flag& operator=(const atomic_flag&) = delete;
  atomic_flag& operator=(const atomic_flag&) volatile = delete;
} atomic_flag;
```

Constructor (C++11):

atomic_flag::atomic_flag (C++11):
Purpose

Constructs an object of type `atomic_flag`.

Prototypes

atomic_flag() noexcept = default;

atomic_flag(const atomic_flag&) = delete;

Usage

The `atomic_flag() noexcept = default;` constructor leaves an `atomic_flag` object with unspecified state.

The `atomic_flag( const atomic_flag& ) = delete;` constructor specifies that the `atomic_flag` object cannot be copy-constructed (copied).

atomic_flag::clear (C++11):
Purpose

Atomically sets the value pointed to by *this* to false.

Memory is affected according to the value of *order*.

Prototypes

void clear(memory_order *order* = memory_order_seq_cst) volatile noexcept;

void clear(memory_order *order* = memory_order_seq_cst) noexcept;

Parameter

order
The memory order constraints to enforce. The *order* parameter cannot be `memory_order_acquire` or `memory_order_acq_rel`.

atomic_flag::test_and_set (C++11):
Purpose

Atomically sets the value pointed to by *this* to true.

Memory is affected according to the value of *order*.

Prototypes

bool atomic_flag::test_and_set(memory_order *order* = memory_order_seq_cst) volatile noexcept;

bool atomic_flag::test_and_set(memory_order *order* = memory_order_seq_cst) noexcept;

Parameter

order
The memory order constraints to enforce.

Return value

Returns the value of the object immediately before the update.

atomic_flag_test_and_set (C++11)
Purpose

Atomically sets a flag to true.

Prototypes

bool atomic_flag_test_and_set(volatile atomic_flag *_object_) noexcept;

bool atomic_flag_test_and_set(atomic_flag *_object_) noexcept;

Parameter

object
A pointer that points to the flag to be set.

Return value

Returns the value of the flag immediately before the update.

atomic_flag_test_and_set_explicit (C++11)
Purpose

Atomically sets a flag to true.

Prototypes

bool atomic_flag_test_and_set_explicit(volatile atomic_flag *_object_, memory_order _order_) noexcept;

bool atomic_flag_test_and_set_explicit(atomic_flag *_object_, memory_order _order_) noexcept;

Parameters

object
A pointer that points to the flag to be set.

order
The memory order constraints to enforce.

Return value

Returns the value of the flag immediately before the update.

atomic_flag_clear (C++11)
Purpose

Atomically sets a flag to false.

Prototypes

void atomic_flag_clear(volatile atomic_flag *_object_) noexcept;

void atomic_flag_clear(atomic_flag *_object_) noexcept;

Parameter

object
A pointer that points to the flag to be set.

atomic_flag_clear_explicit (C++11)
Purpose

Atomically sets a flag to false.

Memory is affected according to the value of *order*.

Prototypes

void atomic_flag_clear_explicit(volatile atomic_flag **object*, memory_order *order*) noexcept;

void atomic_flag_clear_explicit(atomic_flag **object*, memory_order *order*) noexcept;

Parameters

object
A pointer that points to the flag to be set. The *order* parameter cannot be `memory_order_acquire` or `memory_order_acq_rel`.

order
The memory order constraints to enforce.

Initialization (C++11)

atomic_init (C++11)
Purpose

Initializes an atomic object.

Prototypes

void atomic_init(volatile *A *object*, *C desired*) noexcept;

void atomic_init(*A *object*, *C desired*) noexcept;

Parameters

A One of the atomic types.

C The corresponding non-atomic type of *A*.

object
A pointer that points to an atomic object to be initialized.

desired
The value to initialize the atomic object.

ATOMIC_VAR_INIT (C++11)
The following macro expands to an expression that you can use to initialize an atomic variable that is initialization-compatible with *value*. The initial value of an atomic object of automatic storage duration that is not initialized using this macro is undefined.

```
#define ATOMIC_VAR_INIT(value)
```

Concurrent accesses to the variable that is being initialized, even by an atomic operation, constitute a data race.

ATOMIC_FLAG_INIT (C++11)

You must define the `ATOMIC_FLAG_INIT` macro in such a way that it can be used to initialize an object of type `atomic_flag` to the clear state.

```
#define ATOMIC_FLAG_INIT value
```

For a static-duration object, the initialization must be static.

Memory synchronization order (C++11)

kill_dependency (C++11)

Purpose

Ends a dependency chain.

Prototype

template <class T> T kill_dependency(T y) noexcept;

Parameter

y An expression whose return value is to be removed from a dependency tree.

Return value

Returns *y*.

atomic_thread_fence (C++11)

Purpose

Establishes a memory synchronization order of non-atomic and relaxed atomic accesses without an associated atomic operation.

Prototype

extern "C" void atomic_thread_fence (memory_order *order*) noexcept;

Parameter

order
The memory order that is executed by this fence.

Usage

The function has the following effects depending on the value of *order*:

- It has no effects if the value of *order* is `memory_order_relaxed`.
- It is an acquire fence if the value of *order* is `memory_order_acquire` or `memory_order_consume`.
- It is a release fence if the value of *order* is `memory_order_release`.
- It is both an acquire fence and a release fence if the value of *order* is `memory_order_acq_rel`.
- It is a sequentially consistent acquire and release fence if the value of *order* is `memory_order_seq_cst`.

atomic_signal_fence (C++11)
Purpose

Establishes memory synchronization order of non-atomic and relaxed atomic accesses between a thread and a signal handler executed on the same thread.

Prototype

extern "C" void atomic_signal_fence(memory_order *order*) noexcept;

Parameter

order
The memory order that is executed by this fence.

Usage

This function is equivalent to `atomic_thread_fence`, except that the resulting order constraints are established only between a thread and a signal handler executed in the same thread.

You can use `atomic_signal_fence` to specify the order in which actions performed by the thread become visible to the signal handler. Memory is affected according to the value of *order*.

Constexpr functions (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

A non-constructor function that is declared with a `constexpr` specifier is a `constexpr` function. A `constexpr` function is a function that can be invoked within a constant expression.

A `constexpr` function must satisfy the following conditions:

- It is not virtual.
- Its return type is a literal type.
- Each of its parameters must be of a literal type.
- When initializing the return value, each constructor call and implicit conversion is valid in a constant expression.
- Its function body is `= delete` or `= default`; otherwise, its function body must contain only the following statements:
  - `null` statements
  - `static_assert` declarations
  - `typedef` declarations that do not define classes or enumerations
  - `using` directives
  - `using` declarations

– One return statement

When a nonstatic member function that is not a constructor is declared with the constexpr specifier, that member function is constant, and the constexpr specifier has no other effect on the function type. The class of which that function is a member must be a literal type.

The following examples demonstrate the usage of constexpr functions:

```
const int array_size1 (int x) {
  return x+1;
}
// Error, constant expression required in array declaration
int array[array_size1(10)];

constexpr int array_size2 (int x) {
  return x+1;
}
// OK, constexpr functions can be evaluated at compile time
// and used in contexts that require constant expressions.
int array[array_size2(10)];

struct S {
  S() { }
  constexpr S(int) { }
  constexpr virtual int f() {  // Error, f must not be virtual.
    return 55;
  }
};

struct NL {
  ~NL() { }  // The user-provided destructor (even if it is trivial)
             // makes the type a non-literal type.
};

constexpr NL f1() {  // Error, return type of f1 must be a literal type.
  return NL();
}

constexpr int f2(NL) {  // Error, the parameter type NL is not a literal type.
  return 55;
}

constexpr S f3() {
  return S();
}

enum { val = f3() };  // Error, initialization of the return value in f3()
                      // uses a non-constexpr constructor.

constexpr void f4(int x) {  // Error, return type should not be void.
  return;
}

constexpr int f5(int x) { // Error, function body contains more than return statement.
  if (x<0)
    x = -x;
  return x;
}
```

When a function template is declared as a constexpr function, if the instantiation results in a function that does not satisfy the requirements of a constexpr function, the constexpr specifier is ignored. For example:

```
template <class C> constexpr NL f6(C c) {  // OK, the constexpr specifier ignored
  return NL();
}
void g() {
  f6(55);  // OK, not used in a constant expression
}
```

A call to a `constexpr` function produces the same result as a call to an equivalent non-`constexpr` function in all respects, except that a call to a `constexpr` function can appear in a constant expression.

A `constexpr` function is implicitly inline.

The `main` function cannot be declared with the `constexpr` specifier.

Related reference:

"The constexpr specifier (C++11)" on page 83

"Generalized constant expressions (C++11)" on page 149

Chapter 9. Namespaces (C++ only)

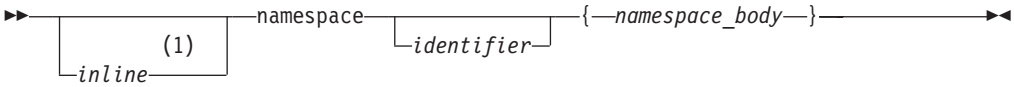
A *namespace* is an optionally named scope. You declare names inside a namespace as you would for a class or an enumeration. You can access names declared inside a namespace the same way you access a nested class name by using the scope resolution (`::`) operator. However namespaces do not have the additional 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.

IBM To be compatible with GNU C, the compiler supports the visibility attribute to namespaces. IBM

Defining namespaces

In order to uniquely identify a namespace, use the **namespace** keyword.

Namespace syntax



Notes:

1 This syntax is valid only at the C++11 language level.

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 reference:

"Unnamed namespaces" on page 319

"Inline namespace definitions (C++11)" on page 324

Declaring namespaces

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 reference:

"File/global scope" on page 3

Creating a namespace alias

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 compile-time 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 compile-time error will result.

Related reference:

"File/global scope" on page 3

Creating an alias for a nested namespace

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.

Related reference:

"Creating a namespace alias" on page 317

Extending namespaces

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

You can overload functions across namespaces. For example:

```
// Original X.h:
int f(int);

// Original Y.h:
int f(char);

// Original program.c:
#include "X.h"
#include "Y.h"

int main(){
  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;

int main(){
  f('a');   // calls f() from Y.h
}
```

In `program.c`, the main function 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 reference:

Chapter 10, “Overloading (C++ only),” on page 327

Unnamed namespaces

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 reference:

"Program linkage" on page 7

"Internal linkage" on page 7

Namespace member definitions

A namespace can define its own members within itself or externally using explicit qualification. See the following 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.

See the following 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

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 compile-time 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 reference:

"Friends" on page 373

The using directive

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.

Using directive syntax

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 with some exceptions.

For unqualified name lookup, 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 acts 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 compile-time error, because function `f()` does not know which `i` to call; `i` from namespace `A`, or `i` from namespace `B`.

Related reference:

"The using declaration and class members" on page 388

"Inline namespace definitions (C++11)" on page 324

The using declaration and namespaces

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.

using declaration syntax

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.

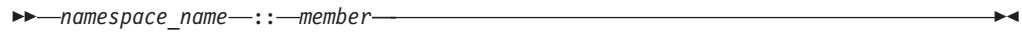
Related reference:

"The `using` declaration and class members" on page 388

Explicit access

To explicitly qualify a member of a namespace, use the namespace identifier with a :: scope resolution operator.

Explicit access qualification syntax



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 reference:

"Scope resolution operator :: (C++ only)" on page 148

Inline namespace definitions (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

Inline namespace definitions are namespace definitions with an initial `inline` keyword. A namespace so defined is an inline namespace. You can define and specialize members of an inline namespace as if they were also members of the enclosing namespace.

Inline namespace definitions syntax

When an inline namespace is defined, a `using` directive is implicitly inserted into its enclosing namespace. While looking up a qualified name through the enclosing namespace, members of the inline namespace are brought in and found by the implicit `using` directive, even if that name is declared in the enclosing namespace.

For example, if you compile the following code with USE_INLINE_B defined, the output of the resulting executable is 1; otherwise, the output is 2.

```
namespace A {
#if USE_INLINE_B
   inline
#endif
   namespace B {
      int foo(bool) { return 1; }
   }
   int foo(int) { return 2; }
}

int main(void) {
   return A::foo(true);
}
```

The properties of inline namespace definitions are transitive; that is, you can use members of an inline namespace as if they were also members of any namespace in its enclosing namespace set, which consists of the innermost non-inline namespace enclosing the inline namespace, together with any intervening inline namespaces. For example:

```
namespace L {
   inline namespace M {
      inline namespace N {
        / *...* /
      }
   }
}
```

In this example, a namespace L contains an inline namespace M, which in turn contains another inline namespace N. The members of N can also be used as if they were members of the namespaces in its enclosing namespace set, i.e., L and M.

Notes:

- Do not declare the namespace `std`, which is used for the C++ standard library, as an inline namespace.
- Do not declare a namespace to be an inline namespace if it is not inline in its first definition.
- You can declare an unnamed namespace as an inline namespace.

Using inline namespace definitions in explicit instantiation and specialization

You can explicitly instantiate or specialize each member of an inline namespace as if it were a member of its enclosing namespace. Name lookup for the primary template of an explicit instantiation or specialization in a namespace, for example M, considers the inline namespaces whose enclosing namespace set includes M. For example:

```
namespace L {
   inline namespace M {
       template <typename T> class C;
   }

   template <typename T> void f(T) { /*...*/ };
}

struct X { /*...*/ };
namespace L {
  template<> class C<X> { /*...*/ };    //template specialization
}

int main()
{
  L::C<X> r;
  f(r);  // fine, L is an associated namespace of C
}
```

In this example, `M` is an inline namespace of its enclosing namespace `L`, class `C` is a member of inline namespace `M`, so `L` is an associated namespace of class `C`.

The following rules apply when you use inline namespace definitions in explicit instantiation and specialization:

- An explicit instantiation must be in an enclosing namespace of the primary template if the template name is qualified; otherwise, it must be in the nearest enclosing namespace of the primary template or a namespace in the enclosing namespace set.
- An explicit specialization declaration must first be declared in the namespace scope of the nearest enclosing namespace of the primary template, or a namespace in the enclosing namespace set. If the declaration is not a definition, it may be defined later in any enclosing namespace.

Using inline namespace definitions in library versioning

With inline namespace definitions, you can provide a common source interface for a library with several implementations, and a user of the library can choose one

implementation to be associated with the common interface. The following example demonstrates the use of inline namespace in library versioning with explicit specialization.

```
//foo.h
#ifndef SOME_LIBRARY_FOO_H_
   #define SOME_LIBRARY_FOO_H_
   namespace SomeLibrary
   {
      #ifdef SOME_LIBRARY_USE_VERSION_2_
         inline namespace version_2 { }
      #else
         inline namespace version_1 { }
      #endif
      namespace version_1 {
         template <typename T> int foo(T a) {return 1;}
      }
      namespace version_2 {
         template <typename T> int foo(T a) {return 2;}
      }
   }
#endif

//myFooCaller.C
#include <Foo.h>
#include <iostream>

struct MyIntWrapper { int x;};

//Specialize SomeLibrary::foo()
//Should specialize the correct version of foo()

namespace SomeLibrary {
   template <> int foo(MyIntWrapper a) { return a.x;}
}

int main(void) {
   using namespace SomeLibrary;
   MyIntWrapper intWrap = { 4 };
   std::cout << foo(intWrap) + foo(1.0) << std::endl;
}
```

If you compile this example with SOME_LIBRARY_USE_VERSION_2_ defined, the output of the resulting executable is 6; otherwise, the output is 5. If the function call, `foo(intWrap)`, is qualified with one of the inline namespaces, then you need to ensure that the explicit specialization is effective.

Related reference:

"Defining namespaces" on page 317

"Extending namespaces" on page 318

"The using directive" on page 322

"The `using` declaration and namespaces" on page 322

"Explicit instantiation" on page 456

"Explicit specialization" on page 460

"C++11 compatibility" on page 640

Chapter 10. Overloading (C++ only)

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. Overloaded functions and operators are described in "Overloading functions" and "Overloading operators" on page 329, respectively.

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*, as described in "Overload resolution" on page 337.

Overloading functions

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
```

Related reference:

"Restrictions on overloaded functions"

"Derivation" on page 383

Restrictions on overloaded functions

You cannot overload the following function declarations even 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 use 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 use 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, see the following declarations of the same function:

  ```
  void f(char*);
  void f(char[10]);
  ```

  The first array dimension is insignificant when differentiating parameters; all other array dimensions are significant. For example, see the following declarations of the same function:

  ```
  void g(char(*)[20]);
  void g(char[5][20]);
  ```

  The following two declarations are *not* equivalent:

  ```
  void g(char(*)[20]);
  void 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, see the following 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, see the following 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, see the following 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.

Related reference:

"The `using` declaration and namespaces" on page 322

"typedef definitions" on page 74

"Type qualifiers" on page 85

"Language linkage (C++ only)" on page 9

Overloading operators

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.

CCNX12B

```
// 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.

Overloading unary operators

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;
}
```

See 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 reference:

"Unary expressions" on page 151

Overloading increment and decrement operators

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 reference:

"Increment operator ++" on page 151

"Decrement operator --" on page 152

Overloading binary operators

You overload a binary 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@(U)
```

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 reference:

"Binary expressions" on page 160

Overloading assignments

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 reference:

"Copy assignment operators" on page 430

"Assignment operators" on page 161

Overloading function calls

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 reference:

"Function call expressions" on page 149

Overloading subscripting

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;
  }
}
```

See 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)`.

Overloading class member access

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 reference:

"Arrow operator ->" on page 150

Overload resolution

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++ runtime environment 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 best viable function is the one whose parameters all have either better or equal-ranked implicit conversion sequences than all of the other viable functions. The compiler will not allow a program in which the compiler was able to find more than one best viable function. Implicit conversion sequences are described in more detail in "Implicit conversion sequences" on page 339.

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

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 tries to determine an implicit conversion sequence for each argument. It then categorizes each implicit conversion sequence in one of three categories and ranks them depending on the category. The compiler does 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" on page 340

Note: Two standard conversion sequences or two user-defined conversion sequences might have different ranks.

Standard conversion sequences

Standard conversion sequences are categorized in one of three ranks. The ranks are listed in order from highest to lowest:

- 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 lowest-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 both `A` and `B` 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.

Ranking implicit conversion sequences

Ranking standard conversion sequences

Suppose S1 and S2 are two standard conversion sequences. The compiler checks whether S1 and S2 satisfy the following conditions in sequence. If one of the conditions is satisfied, S1 is a better standard conversion sequence than S2.

1. S2 involves a qualification conversion, but S1 does not involve qualification conversions. See Example 1.
2. The rank of S1 is higher than the rank of S2. See Example 2.
3. Both S1 and S2 involve qualification conversions. T1 is the target type of S1, and T2 of S2. T2 is more cv-qualified than T1. See Example 3.
4. C++11 S1 and S2 are reference bindings to an rvalue, and neither of them refers to the implicit object parameter of a nonstatic member function. S1 binds an rvalue reference and S2 binds an lvalue reference. See Example 4. C++11
5. C++11 S1 and S2 are reference bindings. S1 binds an lvalue reference to a function lvalue, and S2 binds an rvalue reference to a function lvalue. See Example 5. C++11
6. S1 and S2 are reference bindings. T1 is the target type referred by S1, and T2 by S2. T1 and T2 differ only in top-level cv-qualifiers where T2 is more cv-qualified than T1. See Example 6.

If two standard conversion sequences S1 and S2 have the same rank, S1 is a better standard conversion sequence than S2 if one of the following conditions is satisfied:

- S1 converts a pointer, a pointer to member, or a null pointer, and S2 does not. See Example 7.
- Class A is a parent class of class B. S1 is a conversion from B* to A*, and S2 is a conversion from B* to void*; or S1 is a conversion from A* to void*, and S2 is a conversion from B* to void*. See Example 8.
- Class A is a parent class of class B, and class B is a parent class of class C. One of the following conditions is satisfied:
  - S1 is a conversion from C* to B*, and S2 is a conversion from C* to A*.
  - S1 binds an expression of type C to a reference of type B&, and S2 binds an expression of type C to a reference of type A&.
  - S1 is a conversion from A::* to B::*, and S2 is a conversion from A::* to C::*.
  - S1 is a conversion from C to B, and S2 is a conversion from C to A.
  - S1 is a conversion from B* to A*, and S2 is a conversion from C* to A*. See Example 9.
  - S1 binds an expression of type B to type A&, and S2 binds an expression of type C to type A&.
  - S1 is a conversion from B::* to C::*, and S2 is a conversion from A::* to C::*.
  - S1 is a conversion from B to A, and S2 is a conversion from C to A.

Example 1

```
void f(int*);        // #1 function
void f(const int*);  // #2 function

void test() {
   // The compiler calls #1 function
   f(static_cast<int*>(0));
}
```

In this example, for the call of `f(static_cast<int*>(0))`, the standard conversion sequence `S1` of the `f(int*)` function is from `int*` to `int*`. The standard conversion sequence `S2` of the `f(const int*)` function is from `int*` to `const int*`. `S2` involves a qualification conversion, but `S1` does not, so `S1` is a better standard conversion sequence than `S2`.

Example 2

```
struct A { };
struct B : A { };

void f(const B*);  // #1 function
void f(A*);        // #2 function

void test() {
   // The compiler calls #1 function
   f(static_cast<B*>(0));
}

struct A1 *g(int);    // #3 function
struct A2 *g(short);  // #4 function

void test2() {
  // The compiler calls #4 function
  A2* a2 = g(static_cast<short>(0));

  // The compiler calls #3 function
  A1* a1 = g('\0');
}
```

In this example, for the call of `f(static_cast<B*>(0))`, the standard conversion sequence of the `f(const B*)` function is an exact match, and the standard conversion sequence of the `f(A*)` function is a conversion. The rank of exact match is higher than that of conversion, so `f(const B*)` is chosen by overload resolution. Similarly, for the call of `g(static_cast<short>(0))`, the standard conversion sequence of the `g(short)` function is an exact match, and the standard conversion sequence of the `g(int)` function is a promotion. The `g(short)` function is called because the rank of exact match is higher than that of promotion. For the call of `g('\0')`, the standard conversion sequence of the `g(short)` function is a conversion, and the standard conversion sequence of the `g(int)` function is a promotion, `g(int)` is called in this case because the rank of promotion is higher than that of conversion.

Example 3

```
struct A { };
struct B : A { };
void g(const A*);           // #1 function
void g(const volatile A*);  // #2 function
void test2() {
   // The compiler calls #1 function
   g(static_cast<B*>(0));
}
```

In this example, for the call of `g(static_cast<B*>(0))`, the standard conversion sequence `S1` of the `g(const A*)` function is from `B*` to `const A*`. The standard conversion sequence `S2` of the `g(const volatile A*)` function is from `B*` to `const volatile A*`. Both `S1` and `S2` involve qualification conversions, and `const volatile A*` is more cv-qualified than `const A*`, so `S1` is a better standard conversion sequence than `S2`.

C++11

Example 4

```
double f1();
int g(const double&&);  // #1 function
int g(const double&);   // #2 function

// The compiler calls #1 function
int i = g(f1());

struct A {
   int operator+(int);
};

int operator+(A &&, int);

A &&f2();

void test() {
   f2() + 0;  // error
}
```

In this example, for the call of `g(f1())`, the standard conversion sequence of the `g(const double&)` function binds an lvalue reference, and the standard conversion sequence for `g(const double&&)` binds an rvalue reference. Neither of these two standard conversion sequences refers to the implicit object parameter of a nonstatic member function. The `g(const double&&)` function is called because its standard conversion sequence is a better one. For the expression `f2() + 0`, the class member candidate involves a reference binding of the implicit object parameter of a nonstatic member function and hence cannot be ordered with respect to the namespace scope candidate.

Example 5

```
double f();
int g(double(&&)());  // #1 function
int g(double(&)());   // #2 function

// The compiler calls #2 function
int i = g(f)
```

In this example, for the call of `g(f)`, the standard conversion sequence of the `g(double(&)())` function binds an lvalue reference to a function lvalue, and the standard conversion sequence of the `g(double(&&)())` function binds an rvalue reference to a function lvalue. The `g(double(&)())` function is called because its standard conversion sequence is a better one.

C++11

Example 6

```
void f(A&);        // #1 function
void f(const A&);  // #2 function
void test() {
```

```
   A a;
   // The compiler calls #1 function
   f(a);
}
```

In this example, for the call of `f(a)`, the standard conversion sequence `S1` of the `f(A&)` function binds an lvalue reference `A&` to `a`, and the standard conversion sequence `S2` of the `f(const A&)` function binds a `const` lvalue reference `const A&` to `a`. Because `const A` and `A` are the same except for top-level cv-qualifers, and `const A` is more cv-qualified than `A`, `S1` is a better standard conversion sequence than `S2`.

Example 7

```
void f(void*);  // #1 function
void f(bool);   // #2 function
void test() {
   // The compiler calls #1 function
   f(static_cast<int*>(0));
}
```

In this example, for the call of `f(static_cast<int*>(0))`, the standard conversion sequence `S1` of the `f(void*)` function is from `int*` to `void*`, and the standard conversion sequence `S2` of the `f(bool)` function is from `int*` to `bool`. `S1` and `S2` have the same rank. However, because `S1` does not convert a pointer, a pointer to member, or a null pointer to a `bool`, and `S2` converts a pointer to a `bool`, `S1` is a better standard conversion sequence than `S2`.

Example 8

```
//
void f(void*);      // #1 function
void f(struct A*);  // #2 function
struct A { };
struct B : A { };
void test() {
  // The compiler calls #2 function
  f(static_cast<B*>(0));
}
```

In this example, for the call of `f(static_cast<B*>(0))`, the standard conversion sequence of the `f(void*)` function is from `B*` to `void*`, and the standard conversion sequence of the `f(struct A*)` is from `B*` to `A*`. The `f(struct A*)` is called because its standard conversion sequence is a better one.

Example 9

```
void f(struct A *);
struct A { };
struct B : A { };
struct C : B { };
struct S {
   operator B*();
   operator C*();
}
void test() {
   // calls S::operator B*()
   f(S());
}
```

In this example, for the call of `f(S())`, the standard conversion sequence is from `S()` to `A*`, and the structure `S` has two conversion operators. The operator function `operator B* ()` is called, because the conversion from `B*` to `A*` is better than from `C*` to `A*`.

Ranking user-defined conversion sequences

Suppose U1 and U2 are two user-defined conversion sequences. U1 and U2 use the same user-defined conversion function, constructor, or aggregate initialization. U1 is a better user-defined conversion sequence than U2 if the second standard conversion sequence of U1 is better than that of U2. See Example 10:

Example 10

```
void f(void*);  // #1 function
void f(bool);   // #2 function
struct A {
   operator int*();
}
void test() {
   // The compiler calls #1 function
   f(A());
}
```

In this example, for the call of f(A()), the user-defined conversion sequence U1 of the f(void*) function is from A to void*. The user-defined conversion sequence U2 of the f(bool) function is from A to bool. U1 and U2 use the same user-defined conversion from A to int*. The standard conversion sequence from int* to void* is better than the standard conversion sequence from int* to bool, so U1 is a better user-defined conversion sequence than U2.

Related reference:

"Lvalue-to-rvalue conversions" on page 137

"Pointer conversions" on page 137

"Integral conversions" on page 130

"Floating-point conversions" on page 130

"Boolean conversions" on page 130

"Lvalues and rvalues" on page 141

"References (C++ only)" on page 107

Resolving addresses of overloaded functions

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 reference:

"Pointers to functions" on page 257

"Pointers to members" on page 362

"Function templates" on page 446

"Explicit specialization" on page 460

Chapter 11. Classes (C++ only)

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.

Related reference:

"Structures and unions" on page 60

Chapter 12, "Class members and friends (C++ only)," on page 357

Chapter 13, "Inheritance (C++ only)," on page 381

Chapter 10, "Overloading (C++ only)," on page 327

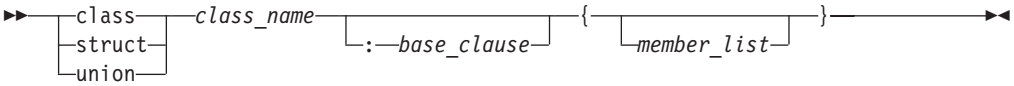
"Virtual functions" on page 399

Declaring class types

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.

Class specifier syntax



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.

Related reference:

"Class member lists" on page 357

"Derivation" on page 383

Using class objects

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
}
```

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.

Objects of class types that are not copy restricted can be assigned, passed as arguments to functions, and returned by functions.

Related reference:

"Structures and unions" on page 60

"References (C++ only)" on page 107

"Scope of class names" on page 351

Classes and structures

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`:

CCNX10C

```
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`:

CCNX10D

```
#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;
}
```

See the output of the above example:

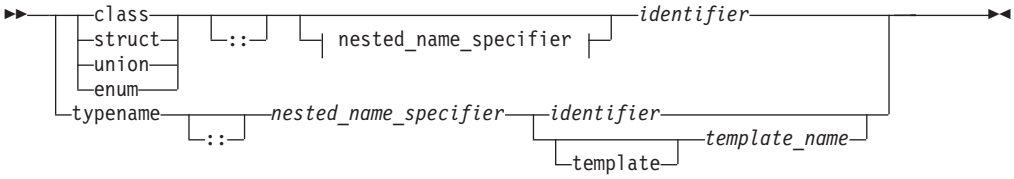
```
Here are a and b: 0 1
```

Scope of class names

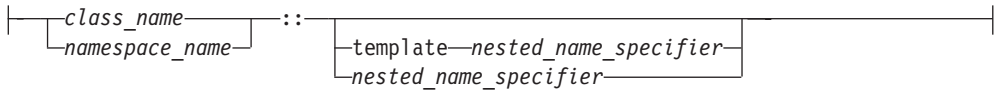
A class declaration introduces the class name into the scope where it is declared. Any class, object, function or other declaration of that name in an enclosing scope is hidden.

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*:

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.

Related reference:

"Incomplete class declarations" on page 352

Incomplete class declarations

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 as 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 reference:

"Class member lists" on page 357

Nested classes

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.

Related reference:

"Class scope (C++ only)" on page 4

"Scope of class names" on page 351

"Member access" on page 371

"Static members" on page 366

Local classes

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 reference:

"Member functions" on page 359

"The inline function specifier" on page 228

Local type names

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 reference:

"Scope" on page 2

"typedef definitions" on page 74

Chapter 12. Class members and friends (C++ only)

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.

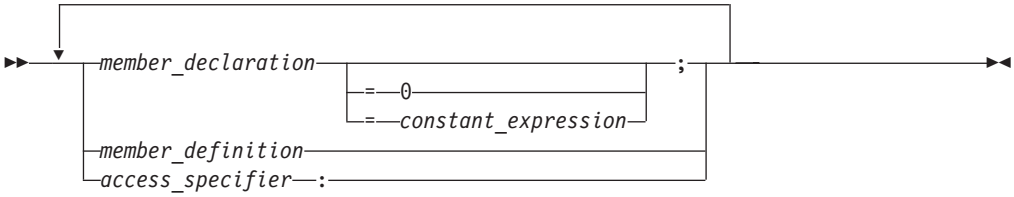
Related reference:

"Inherited member access" on page 386

Class member lists

An optional *member list* declares subobjects called *class members*. Class members can be data, functions, nested types, and enumerators.

Class member list syntax



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.

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

Related reference:

"Declaring class types" on page 347

"Member access" on page 371

"Inherited member access" on page 386

"Static members" on page 366

Data members

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 &xlref;
      X &&xrref;
      static X xcount;
      X xfunc(X);
};
```

Related reference:

"Member access" on page 371

"Inherited member access" on page 386

"Static members" on page 366

Member functions

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*.

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;
};
```

C++11

You can use trailing return types for member functions, including those that have complicated return types. For more information, see "Trailing return type (C++11)" on page 239.

C++11

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* Y::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 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.

Constant and volatile member functions

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 reference:

"The this pointer" on page 363

Virtual member functions

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 reference:

"Virtual functions" on page 399

"Abstract classes" on page 404

Special member functions

Special member functions are used to create, destroy, initialize, convert, and copy class objects. These include the following elements:

- Default constructors
- Destructors
- Copy constructors
- Copy assignment operators

For full descriptions of these functions, see Chapter 14, “Special member functions (C++ only),” on page 407.

Member scope

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.

CCNX11A

```
#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.

Pointers to members

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`.

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.

The this pointer

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()` stores 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*`. 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*`.

A `const this` pointer can by used only with `const` member functions. Data members of the class will be constant 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.

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`:

CCNX11C

```
#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 <iostream>
using namespace std;
 	
struct X {
private:
  int len;
  char *ptr;
public:
  int GetLen() {
  return len;
  }
  char * GetPtr() {
  return ptr;
  }
  X& Set(char *);
  X& Cat(char *);
  X& Copy(X&);
  void Print();
};
 	
X& X::Set(char *pc) {
  len = strlen(pc);
  ptr = new char[len];
  strcpy(ptr, pc);
  return *this;
}
 	
X& X::Cat(char *pc) {
  len += strlen(pc);
  strcat(ptr,pc);
  return *this;
}
 	
X& X::Copy(X& x) {
  Set(x.GetPtr());
  return *this;
}
 	
void X::Print() {
  cout << ptr << endl;
}
 	
int main() {
  X xobj1;
  xobj1.Set("abcd")
  .Cat("efgh");
 	
  xobj1.Print();
  X xobj2;
  xobj2.Copy(xobj1)
  .Cat("ijkl");
 	
  xobj2.Print();
}
	#include <string>
	#include <iostream>
	using namespace std;
	
	struct X {
	private:
	  int len;
	  char *ptr;
	public:
	  int GetLen (X* const THIS) {
	  return THIS->len;
	  }
	  char * GetPtr (X* const THIS) {
	  return THIS->ptr;
	  }
	  X& Set(X* const, char *);
	  X& Cat(X* const, char *);
	  X& Copy(X* const, X&);
	  void Print(X* const);
	};
	
	X& X::Set(X* const THIS, char *pc) {
	  THIS->len = strlen(pc);
	  THIS->ptr = new char[THIS->len];
	  strcpy(THIS->ptr, pc);
	  return *THIS;
	}
	
	X& X::Cat(X* const THIS, char *pc) {
	  THIS->len += strlen(pc);
	  strcat(THIS->ptr, pc);
	  return *THIS;
	}
	
	X& X::Copy(X* const THIS, X& x) {
	  THIS->Set(THIS, x.GetPtr(&x));
	  return *THIS;
	}
	
	void X::Print(X* const THIS) {
	  cout << THIS->ptr << endl;
	}
	
	int main() {
	  X xobj1;
	  xobj1.Set(&xobj1 , "abcd")
	  .Cat(&xobj1 , "efgh");
	
	  xobj1.Print(&xobj1);
	  X xobj2;
	  xobj2.Copy(&xobj2 , xobj1)
	  .Cat(&xobj2 , "ijkl");
	
	  xobj2.Print(&xobj2);
	}

Both examples produce the following output:

```
abcdefgh
abcdefghijkl
```

Related reference:

"Overloading assignments" on page 334

"Copy constructors" on page 429

Static members

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 reference:

"Constant and volatile member functions" on page 360

"Class member lists" on page 357

Using the class access operators with static members

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() { }
```

See the following 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.

Related reference:

"The static storage class specifier" on page 49

"Scope resolution operator :: (C++ only)" on page 148

"Dot operator ." on page 150

"Arrow operator ->" on page 150

Static data members

The declaration of a static data member in the member list of a class is not a 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.

Static data members of a class in namespace scope have external linkage. 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`. You cannot declare a static data member as `mutable`.

You can only have one definition of a static member in a program. Unnamed classes, classes contained within unnamed classes, and local classes cannot have static data members.

Static data members and their initializers can access other static private and protected members of their class. 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();    // error
int C::n = Y::r;      // error
int C::p = y.r;       // error
int C::q = y.f();     // error
```

The initialization of `C::m`, `C::n`, `C::p`, and `C::q` causes errors because the values used to initialize them are private members of class Y which can not be accessed.

If a static data member is of a `const` integral or `const` enumeration type, you can specify a *constant initializer* in the static data member's declaration. This constant initializer must be an integral constant expression.

C++11 A static data member of a literal type can be declared with the `constexpr` specifier in the class definition, and the data member declaration must specify a constant initializer. For example:

```
struct Constants {
  static constexpr int bounds[] = { 42, 56 };
};

float a[Constants::bounds[0]][Constants::bounds[1]];
```

C++11

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.

Related reference:

"External linkage" on page 8

"Member access" on page 371

"Local classes" on page 354

Static member functions

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; // error
  }

};

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:

CCNX11H

```
#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 reference:

"The this pointer" on page 363

Member access

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 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.
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.

Related reference:

"Scope" on page 2

"Class member lists" on page 357

"Inherited member access" on page 386

Friends

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.

CCNX11I

```
#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);
}
```

See 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.

CCNX11J

```
#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);
}
```

See the output of the above example:

```
a is 1
b is 2
```

You cannot define a class in a friend declaration. For example, the compiler does not accept the following code:

```
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. The function must have namespace scope, and the function name must be unqualified. The following example demonstrates this:

```
class A {
  void g();
};

void z() {
  class B {
    friend void f() { }; // error
  };
}

class C {
  friend void A::g() { } // error
  friend void h() { }
};
```

The compiler accepts the definition of `h()`, but not the function definition of `f()` or `g()`.

You cannot declare a friend with a storage class specifier.

C++11

Extended friend declarations

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features

is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.
In the C++11 standard, the *extended friend declarations* feature accepts additional forms of non-function friend declarations.

Note: The syntactic form of extended friend declarations overlaps with the IBM old friend declaration syntax. This section is focused on the differences between the C++11 standard and the previous ISO C++ standard.

With this feature enabled, the class-key is no longer required in the context of friend declarations. This new syntax differs from the C++98 friend class declaration syntax, where the class-key is necessary as part of an elaborated-type-specifier. See the following example:

```
class F;
class G;

class X1 {
  //C++98 friend declarations remain valid in C++11.
  friend class F;

  //Error in C++98 for missing the class-key.
  friend G;
};

class X2 {
  //Error in C++98 for missing the class-key.
  //Error in C++11 for lookup failure (no previous class D declaration).
  friend D;

  friend class D;
};
```

In addition to functions and classes, you can also declare template parameters and basic types as friends. In this case, you cannot use an elaborated-type-specifier in the friend declaration. In the following example, you can declare the template parameter `T` as a friend of class `F`, and you can use the basic type `char` in friend declarations.

```
class C;

template <typename T, typename U> class F {
   //C++11 compiles sucessfully.
   //Error in C++98 for missing the class-key.
   friend T;

   //Error in both C++98 and C++11: a template parameter
   //must not be used in an elaborated type specifier.
   friend class U;

};

F<C> rc;
F<char> Ri;
```

You can also declare `typedef` names as friends, but you still cannot use an elaborated-type-specifier in the friend declaration. The following example demonstrates that the `typedef` name `D` is declared as a friend of class `Base`.

```
class Derived;
typedef Derived D;
```

```
class C;
typedef C Ct;

class Base{
public:
   Base() : x(55) {}

   //C++11 compiles sucessfully.
   //Error in C++98 for missing the class-key.
   friend D;

   //Error in both C++98 and C++11: a typedef name
   //must not be used in an elaborated type specifier.
   friend class Ct;

private:
   int x;
};

struct Derived : public Base {
   int foo() { return this->x; }
};

int main() {
   Derived d;
   return d.foo();
}
```

This feature also introduces a new name lookup rule for friend declarations. If a friend class declaration does not use an elaborated-type-specifier, then the compiler also looks for the entity name in scopes outside the innermost namespace that encloses the friend declaration. Consider the following example:

```
struct T { };

namespace N {
   struct A {
      friend T;
   };
}
```

In this example, if this feature is in effect, the friend declaration statement does not declare a new entity `T`, but looks for `T`. If there is no `T` found, then the compiler issues an error. Consider another example:

```
struct T { };

namespace N {
   struct A {
        friend class T;  //fine, no error
   };
}
```

In this example, the friend declaration statement does not look for `T` outside namespace `N`, nor does it find `::T`. Instead, this statement declares a new class `T` in namespace `N`.

C++11

Related reference:

"Static member functions" on page 369

"The inline function specifier" on page 228

"Local classes" on page 354

"Member access" on page 371
"Inherited member access" on page 386
"C++11 compatibility" on page 640

Friend scope

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 inside 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 friend class name first introduced by a friend declaration is considered to belong to the first nonclass enclosing scope. Until a matching declaration is provided in that scope, the class name is not found by name lookup. For example:

```
namespace A {    //the first nonclass scope
   class B {
        class C {
            friend class D;
        }
   };
};
```

In this example, the first nonclass scope that encloses the friend declaration of class `D` is namespace `A`, so friend class `D` is in the scope of namespace `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.

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; // error
  }
};
```

The compiler does not support 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; // error
  }
};
```

The compiler does not accept 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 name is unqualified, the compiler looks 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 class. You do not have to do so with classes. 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(); // error
    friend void b();
    friend void c(); // error
  };
  ::X moocow;
    X moocow2;
}
```

In the above example, the compiler accepts 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`.
- `X moocow2`: This declaration also creates an object of the nonlocal class `::X`.

The compiler does not accept 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 does not accept this statement.
- `friend void c()`: Since function `c()` has not been declared in the scope of `f()`, the compiler does not accept this statement.

Related reference:

"Scope of class names" on page 351

"Nested classes" on page 352

"Local classes" on page 354

Friend access

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 reference:

"Member access" on page 371

Chapter 13. Inheritance (C++ only)

Inheritance is a mechanism of reusing and extending existing classes without modifying them, thus producing hierarchical relationships between 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. The new class contains 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. See "Multiple inheritance" on page 392 for more detailed information.

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:

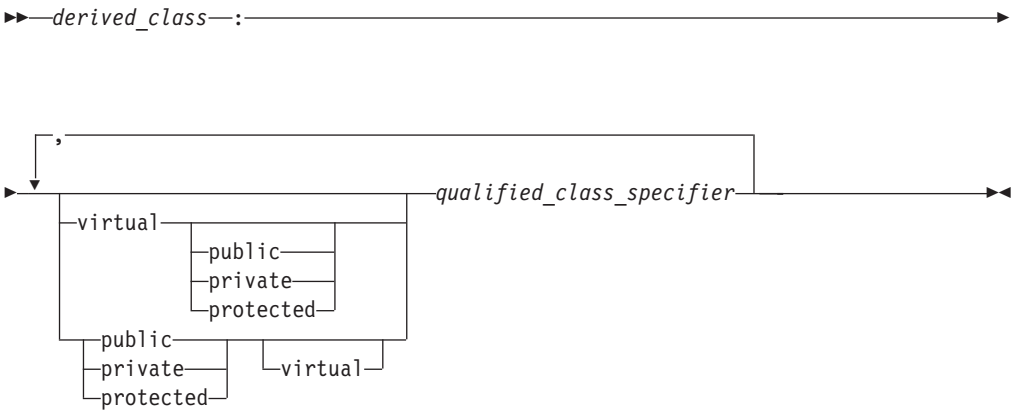
- Overloaded functions are statically bound at compile time.

- 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. They are described in more detail in "Virtual functions" on page 399.

Derivation

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*.

Derived class syntax



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 nonstatic data members of the base class. You can refer to inherited members (base class members) as if they were members of the derived class. The derived class can also add new class members. For example:

CCNX14A

```
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
}
```

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.

The derived class can also declare class members with the same name as existing base class members. You can refer to the base class members by using the `::` (scope resolution) operator. For example:

CCNX14B

```
#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:

CCNX14C

```
#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 has the same name as a base class, the base class name is hidden in the derived class.

Related reference:

"Virtual base classes" on page 393

"Inherited member access"

"Incomplete class declarations" on page 352

"Scope resolution operator :: (C++ only)" on page 148

Inherited member access

The following sections discuss the access rules affecting a protected nonstatic base class member and how to declare a derived class using an access specifier:

- "Protected members"
- "Access control of base class members" on page 387

Related reference:

"Member access" on page 371

Protected members

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`.

Access control of base class members

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`.

See the following example:

```
struct B{
};

class D : B {        // private derivation
};

struct E : B{        // public 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 reference:

"Member access" on page 371

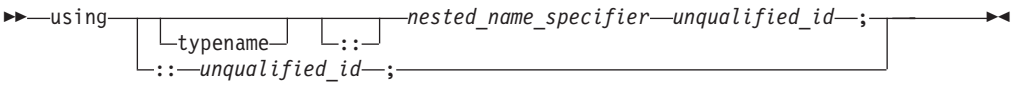
"Member scope" on page 361

The `using` declaration and class members

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 member functions from base and derived classes, or you want to change the access of a class member.

using declaration syntax



A `using` declaration in a class `A` may name one of the following options:

- 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.

Related reference:

"Scope of class names" on page 351

"The `using` declaration and namespaces" on page 322

Overloading member functions from base and derived classes

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()`.

Suppose that you introduce a function `f` from a base class `A` into 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);
}
```

See the following output of the above example:

```
void B::f(int)
```

You can overload virtual functions with a `using` declaration. For example:

```
#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;
  A* pa = &obj_B;
  pa->f(3);
  pa->f();
}
```

In this example, `B::f(int)` is a virtual function and overrides `A::f(int)` even with the `using A::f;` declaration. The output is as below:

```
void B::f(int)
void A::f()
```

Related reference:

Chapter 10, "Overloading (C++ only)," on page 327

"Name hiding (C++ only)" on page 6

"The using declaration and class members" on page 388

Changing the access of a class member

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`.

Related reference:

"Member access" on page 371

"Inherited member access" on page 386

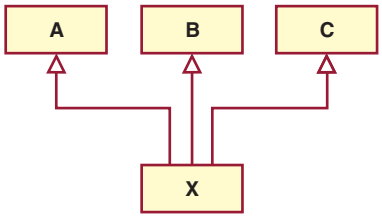
Multiple inheritance

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:

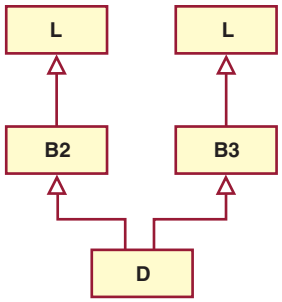


The order of derivation is relevant only to determine the order of default initialization by constructors and cleanup by destructors.

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.
```

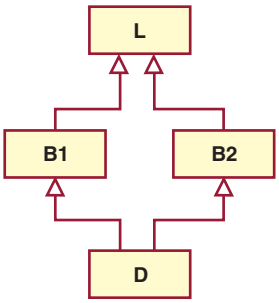
You can also avoid this ambiguity by using the base specifier `virtual` to declare a base class, as described in "Derivation" on page 383.

Virtual base classes

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.

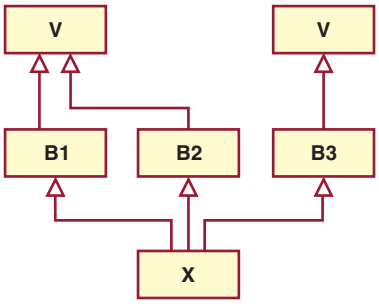
For example:




```
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.

Related reference:

"Derivation" on page 383

Multiple access

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.

Related reference:

"Member access" on page 371

"Protected members" on page 386

"Access control of base class members" on page 387

Ambiguous base classes

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.

CCNX14G

```
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;
   void f() { x = 0; }
};

int main() {
   B b;
   b.f();
}
```

The assignment `x = 0` in function `B::f()` is not ambiguous because the declaration `B::x` has hidden `A::x`.

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 `C::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
}
```

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.

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 reference:

"Scope resolution operator :: (C++ only)" on page 148

"Virtual base classes" on page 393

Virtual functions

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()` is an lvalue reference to an object derived from `A`; it cannot determine whether the argument is an lvalue 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 lvalue reference, but by the type of object that the lvalue 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 {
   virtual 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 pointer or a reference 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`.
- `A::f` returns an lvalue reference if and only if `B::f` returns an lvalue reference.

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.

C++11 A function that has a deleted definition cannot override a function that does not have a deleted definition. Likewise, a function that does not have a deleted definition cannot override a function with a deleted definition. C++11

A virtual function must be one of the following:

- Defined
- Declared pure
- Defined and declared pure

A class containing one or more pure virtual member functions is called an *abstract class*.

Related reference:

"Abstract classes" on page 404

"Deleted functions" on page 222

Ambiguous virtual function calls

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

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

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::f()`, as this overrides the inherited pure virtual function `AB::f()`. Function `AB::f()` needs to be overridden if you want to avoid the abstraction of `D2`.

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()`).

Related reference:

"Virtual functions" on page 399

"Virtual function access" on page 404

Chapter 14. Special member functions (C++ only)

The default constructors, destructors, copy constructors, and copy assignment operators are *special member functions*. These functions create, destroy, convert, initialize, and copy class objects.

C++11 A special member function is user-provided if it is user-declared but not explicitly defaulted, or deleted on its first declaration. C++11

Overview of constructors and destructors

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 or overload 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 reference:

"new expressions (C++ only)" on page 185

Constructors

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.

Default constructors

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 default parameterless 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 C++11 or explicitly defaulted C++11 constructors of the base classes and nonstatic data members of a class `A` before defining the implicitly declared C++11 or explicitly defaulted C++11 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 declared C++11 or explicitly defaulted C++11 .
- `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.

The following example defines a class with one constructor and two default constructors.

```
class X {
public:
  X();                        // Default constructor with no arguments
  X(int = 0);                 // Default constructor with one default argument
  X(int, int , int = 0);      // Constructor
};
```

Note: C++11 You can declare default constructors as explicitly defaulted functions or deleted functions. For more information, see "Explicitly defaulted functions" on page 221 and "Deleted functions" on page 222. C++11

Related reference:

"Copy constructors" on page 429

Chapter 14, "Special member functions (C++ only)," on page 407

Delegating constructors (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this

standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

Before C++11, common initializations in multiple constructors of the same class could not be concentrated in one place in a robust, maintainable manner. To partially alleviate this problem in the existing C++ programs, you could use *assignment* instead of initialization or add a common initialization function.

With the delegating constructors feature, you can concentrate common initializations and post initializations in one constructor named *target constructor*. Delegating constructors can call the target constructor to do the initialization. A delegating constructor can also be used as the *target constructor* of one or more delegating constructors. You can use this feature to make programs more readable and maintainable.

Delegating constructors and target constructors present the same interface as other constructors. Target constructors do not need special handling to become the target of a delegating constructor. They are selected by overload resolution or template argument deduction. After the target constructor completes execution, the delegating constructor gets control back.

In the following example, `X(int x, int y)` and `X(int x, int y, int z)` both delegate to `X(int x)`. This example demonstrates a typical usage of placing common initializations in a single constructor.

```
#include <cstdio>

struct X {
   const int i;
   X(int x, int y) : X(x+y) { }
   X(int x, int y, int z) : X(x*y*z) {}
   X(int x) : i(x) { }
};

int main(void){
   X var1(55,11);
   X var2(2,4,6);
   std::printf("%d, %d\n", var1.i, var2.i);
   return 0;
}
```

The output of the example is:

```
66,48
```

A delegating constructor can be a target constructor of another delegating constructor, thus forming a delegating chain. The first constructor invoked in the construction of an object is called *principal constructor*. A constructor cannot delegate to itself directly or indirectly. The compiler can detect this violation if the constructors involved in a recursive chain of delegation are all defined in one translation unit. Consider the following example:

```
struct A{
  int x,y;
  A():A(42){}
  A(int x_):A() {x = x_;}
};
```

In the example, there is an infinitely recursive cycle that constructor `A()` delegates to constructor `A(int x_)`, and `A(int x_)` also delegates to `A()`. The compiler issues an error to indicate the violation.

You can use the delegating constructors feature interacting with other existing techniques:

- When several constructors have the same name, name and overload resolution can determine which constructor is the target constructor.
- When using delegating constructors in a template class, the deduction of template parameters works normally.

Related reference:

"C++11 compatibility" on page 640

Constexpr constructors (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

A constructor that is declared with a `constexpr` specifier is a `constexpr` constructor. Previously, only expressions of built-in types could be valid constant expressions. With `constexpr` constructors, objects of user-defined types can be included in valid constant expressions.

Definitions of `constexpr` constructors must satisfy the following requirements:

- The containing class must not have any virtual base classes.
- Each of the parameter types is a literal type.
- Its function body is `= delete` or `= default`; otherwise, it must satisfy the following constraints:
  - It is not a function try block.
  - The compound statement in it must contain only the following statements:
    - `null` statements
    - `static_assert` declarations
    - `typedef` declarations that do not define classes or enumerations
    - `using` directives
    - `using` declarations
- Each nonstatic data member and base class subobject is initialized.
- Each constructor that is used for initializing nonstatic data members and base class subobjects is a `constexpr` constructor.
- Initializers for all nonstatic data members that are not named by a member initializer identifier are constant expressions.
- When initializing data members, all implicit conversions that are involved in the following context must be valid in a constant expression:
  - Calling any constructors
  - Converting any expressions to data member types

The implicitly defined default constructor performs the set of initializations of the class that would be performed by a user-written default constructor for that class with no initializer and an empty compound-statement. If that user-defined default constructor would satisfy the requirements of a constexpr constructor, the implicitly defined default constructor is a constexpr constructor.

A constexpr constructor is implicitly inline.

The following examples demonstrate the usage of constexpr constructors:

```
struct BASE {
};

struct B2 {
  int i;
};

//NL is a non-literal type.
struct NL {
  virtual ~NL() {
  }
};

int i = 11;

struct D1 : public BASE {
  //OK, the implicit default constructor of BASE is a constexpr constructor.
  constexpr D1() : BASE(), mem(55) { }

  //OK, the implicit copy constructor of BASE is a constexpr constructor.
  constexpr D1(const D1& d) : BASE(d), mem(55) { }

  //OK, all reference types are literal types.
  constexpr D1(NL &n) : BASE(), mem(55) { }

  //The conversion operator is not constexpr.
  operator int() const { return 55; }

private:
  int mem;
};

struct D2 : virtual BASE {
  //error, D2 must not have virtual base class.
  constexpr D2() : BASE(), mem(55) { }

private:
  int mem;
};

struct D3 : B2 {
  //error, D3 must not be a function try block.
  constexpr D3(int) try : B2(), mem(55) { } catch(int) { }

  //error, illegal statement is in body.
  constexpr D3(char) : B2(), mem(55) { mem = 55; }

  //error, initializer for mem is not a constant expression.
  constexpr D3(double) : B2(), mem(i) { }

  //error, implicit conversion is not constexpr.
  constexpr D3(const D1 &d) : B2(), mem(d) { }

  //error, parameter NL is a non-literal type.
  constexpr D3(NL) : B2(), mem(55) { }
```

```
private:
  int mem;
};
```

Related reference:

"Generalized constant expressions (C++11)" on page 149

"The constexpr specifier (C++11)" on page 83

"Type specifiers" on page 54

"Explicitly defaulted functions" on page 221

"Deleted functions" on page 222

Explicit initialization with constructors

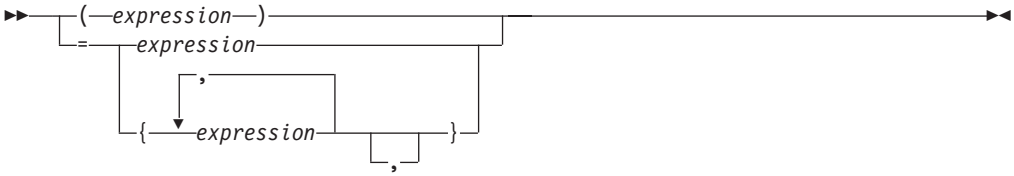
A class object with a constructor must be explicitly initialized or have a default constructor. Except for aggregate initialization, explicit initialization using a constructor is the only way to initialize non-static constant and reference class members.

A class object that has only implicitly declared C++11 or explicitly defaulted C++11 constructors, and has no virtual functions, no private or protected non-static data 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.

Initializer syntax



The following example shows the declaration and use of several constructors that explicitly initialize class objects:

CCNX13A

```
// 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 reference:

"Initializers" on page 108

Chapter 14, "Special member functions (C++ only)," on page 407

Initialization of base classes and members

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:

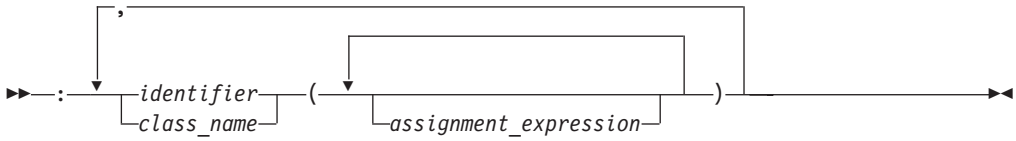
```
complx(double r, double i = 0.0) { re = r; im = i; }
```

Or a constructor can have an *initializer list* within the definition but prior to the constructor 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.

Initializer list syntax



Include the initialization list as part of the constructor 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 output of this example:

```
B1::B1(int)
B2::B2()
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 cases 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 items:

- `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 result:

```
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;
}
```

See 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.

C++11

If the delegating constructors feature is enabled, initialization can only be done within the non-delegating constructor. In other words, a delegating constructor cannot both delegate and initialize. Consider the following example:

```
struct A{
  int x,y;
  A(int x):x(x),y(0){}

  /* the following statement is not allowed */
  A():y(0),A(42) {}
}
```

Constructor `A()` delegates to `A(int x)`, but `A()` also does the initialization, which is not permitted. The compiler issues an error to indicate the violation.

For more information, see "Delegating constructors (C++11)" on page 409

C++11

Related reference:

"The typeid operator (C++ only)" on page 155

"The dynamic_cast operator (C++ only)" on page 182

Constructor execution order for class objects

When a class object is created using constructors, the execution order of constructors is:

1. Constructors of Virtual base classes are executed, in the order that they appear in the base list.
2. Constructors of nonvirtual base classes are executed, in the declaration order.
3. Constructors of class members are executed in the declaration order (regardless of their order in the initialization list).
4. The body of the constructor is executed.

The following example demonstrates these:

```
#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.

C++11

When the construction of a class object is done through a delegating constructor, the body of the delegating constructor is processed after the execution of its target constructor. The rule also applies to the target constructor if the target constructor is another delegating constructor.

Example:

```
#include <cstdio>
using std::printf;

class X{
public:
    int i,j;
    X();
    X(int x);
    X(int x, int y);
    ~X();
}

X::X(int x):i(x),j(23) {printf("X:X(int)\n");}
X::X(int x, int y): X(x+y) { printf("X::X(int,int)\n");}
X::X():X(44,11) {printf("X:X()\n");}
X::~X() {printf("X::~X()\n");}

int main(void){
  X x;
}
```

The output of the example is:

```
X::X(int)
X::X(int,int)
X:X()
X::~X()
```

For more information, see “Delegating constructors (C++11)” on page 409

C++11

Related reference:

“Virtual base classes” on page 393

Destructors

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 C++11 or explicitly defaulted C++11 destructors of the base classes and nonstatic data members of a class `A` before defining the implicitly declared C++11 or explicitly defaulted C++11 destructor of `A`.

A destructor of a class `A` is *trivial* if all the following are true:

- It is implicitly declared C++11 or explicitly defaulted C++11 .
- 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()`).

Note: C++11 You can declare destructors as explicitly defaulted functions or deleted functions. For more information, see "Explicitly defaulted functions" on page 221 and "Deleted functions" on page 222. C++11

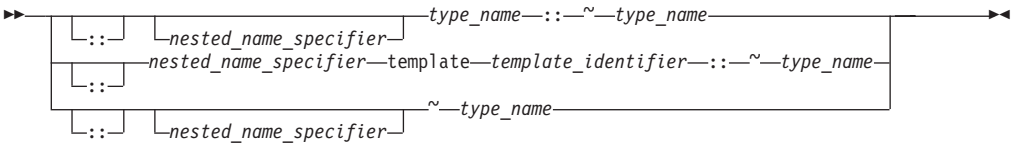
Related reference:

Chapter 14, "Special member functions (C++ only)," on page 407

Pseudo-destructors

A *pseudo-destructor* is a destructor of a nonclass type.

Pseudo-destructor syntax



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 reference:

Chapter 12, "Class members and friends (C++ only)," on page 357

"Scope of class names" on page 351

User-defined conversions

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 constructors
- 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.

Note: C++11 You can declare user-defined conversions as deleted functions. For more information, see "Deleted functions" on page 222. C++11

Conversion constructors

A *conversion constructor* is a single-parameter constructor that is declared without the function specifier `explicit`. The compiler uses conversion constructors to convert objects from the type of the first parameter to the type of the conversion 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 conversion 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 conversion constructor.

Related reference:

"new expressions (C++ only)" on page 185

"The static_cast operator (C++ only)" on page 177

Explicit conversion constructors

The `explicit` function specifier 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 conversion 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 are 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);
```

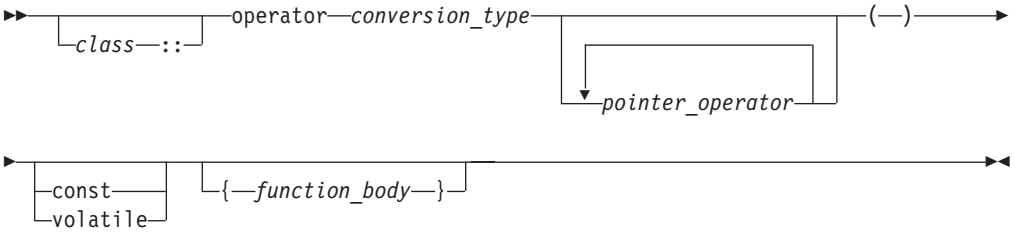
Related reference:

"Conversion constructors" on page 424

Conversion functions

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.

Conversion function syntax



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 reference:

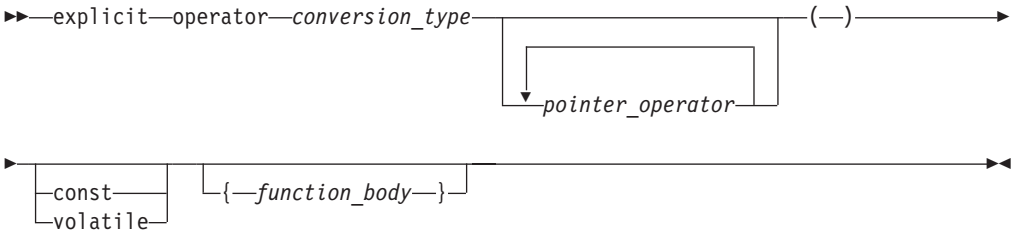
"Conversion constructors" on page 424

Explicit conversion operators (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

You can apply the `explicit` function specifier to the definition of a user-defined conversion function to inhibit unintended implicit conversions from being applied. Such conversion functions are called explicit conversion operators.

Explicit conversion operator syntax



The following example demonstrates both intended and unintended implicit conversions through a user-defined conversion function, which is not qualified with the `explicit` function specifier.

Example 1

```
#include <iostream>

template <class T> struct S {
  operator bool() const;   // conversion function
};

void func(S<int>& s) {
```

```
  // The compiler converts s to the bool type implicitly through
  // the conversion function. This conversion might be intended.
  if (s) { }
}

void bar(S<int>& p1, S<float>& p2) {
  // The compiler converts both p1 and p2 to the bool type implicitly
  // through the conversion function. This conversion might be unintended.
  std::cout << p1+p2 << std::endl;

  // The compiler converts both p1 and p2 to the bool type implicitly
  // through the conversion function and compares results.
  // This conversion might be unintended.
  if (p1==p2) { }
}
```

To inhibit unintended implicit conversions from being applied, you can define an explicit conversion operator by qualifying the conversion function in Example 1 with the `explicit` function specifier:

```
explicit operator bool() const;
```

If you compile the same code as Example 1 but with the explicit conversion operator, the compiler issues error messages for the following statements:

```
// Error: The call does not match any parameter list for "operator+".
std::cout << p1+p2 << std::endl;

// Error: The call does not match any parameter list for "operator==".
if(p1==p2)
```

If you intend to apply the conversion through the explicit conversion operator, you must call the explicit conversion operator explicitly as in the following statements, and then you can get the same results as Example 1.

```
std::cout << bool(p1)+bool(p2) << std::endl;

if(bool(p1)==bool(p2))
```

In contexts where a Boolean value is expected, such as when `&&`, `||`, or the conditional operator is used, or when the condition expression of an `if` statement is evaluated, an explicit `bool` conversion operator can be implicitly invoked. So when you compile Example 1 with the previous explicit conversion operator, the compiler also converts `s` in the `func` function to the `bool` type through the explicit `bool` conversion operator implicitly. Example 2 also demonstrates this:

Example 2

```
struct T {
  explicit operator bool();   //explicit bool conversion operator
};

int main() {
  T t1;
  bool t2;

  // The compiler converts t1 to the bool type through
  // the explicit bool conversion operator implicitly.
  t1 && t2;

  return 0;
}
```

Copy constructors

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&)`.

A program is ill-formed if it includes a class `A` whose copy constructor is implicitly defined C++11 or explicitly defaulted C++11 when one or more of the following conditions 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 C++11 or explicitly defaulted C++11 constructor of a class `A` if you initialize an object of type `A` or an object derived from class `A`.

An implicitly defined C++11 or explicitly defaulted C++11 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.

Note: C++11 You can declare copy constructors as explicitly defaulted functions or deleted functions. For more information, see "Explicitly defaulted functions" on page 221 and "Deleted functions" on page 222. C++11

Related reference:

"Overview of constructors and destructors" on page 407

Chapter 14, "Special member functions (C++ only)," on page 407

Copy assignment operators

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 that is 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;
}
```

See 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 statements 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 an lvalue 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 a copy assignment operator for a class A is implicitly defined C++11 or explicitly defaulted C++11 when one or more of the following conditions 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.

Note: C++11 You can declare copy assignment operators as explicitly defaulted functions or deleted functions. For more information, see "Explicitly defaulted functions" on page 221 and "Deleted functions" on page 222. C++11

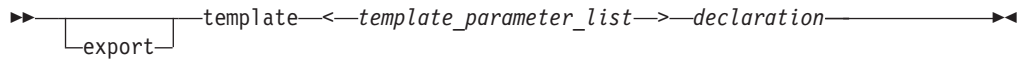
Related reference:

"Assignment operators" on page 161

Chapter 15. Templates (C++ only)

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*, and is described in detail in "Template instantiation" on page 456. This class or function definition generated from a template and a set of template parameters is called a *specialization*, as described in "Template specialization" on page 460.

Template declaration syntax



The compiler accepts and silently ignores the `export` keyword on a template.

The *template_parameter_list* is a comma-separated list of template parameters, which are described in "Template parameters" on page 434.

The *declaration* is one of the following options:

- 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 T> class Key
{
    T k;
    T* kptr;
    int length;
public:
    Key(T);
    // ...
};
```

Suppose the following declarations appear later:

```
Key<int> i;
Key<char*> c;
Key<mytype> m;
```

The compiler would create three instances of `class Key`. The following table shows the definitions of these three class instances 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

There are three kinds of template parameters:

- "Type template parameters"
- "Non-type template parameters"
- "Template template parameters" on page 435

C++11

Template parameter packs can also be a kind of template parameter. For more information, see "Variadic templates (C++11)" on page 468.

C++11

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.

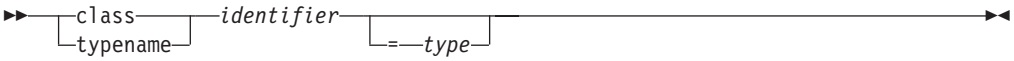
Related reference:

"Type qualifiers" on page 85

"Lvalues and rvalues" on page 141

Type template parameters

Type template parameter declaration syntax



The *identifier* is the name of a type.

Related reference:

"The typename keyword" on page 482

Non-type template parameters

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

- lvalue reference to object or lvalue reference to function
- pointer to member
- C++11 std::nullptr_t C++11

Non-type template parameters that are declared as arrays are converted to pointers, and that are declared as functions are converted to pointers to functions. The following example demonstrates these rules:

```
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 can 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 non-reference template parameters are not lvalues.

Related reference:

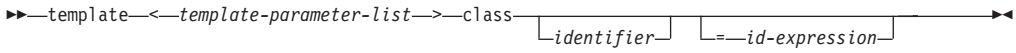
"Type qualifiers" on page 85

"Lvalues and rvalues" on page 141

"References (C++ only)" on page 107

Template template parameters

Template template parameter declaration syntax



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

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. The compiler will not allow the following example:

```
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 { };
```

Naming template parameters as friends (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

In the C++11 standard, the extended friend declarations feature is introduced, with which you can declare template parameters as friends. This makes friend declarations inside templates easier to use.

If a friend declaration resolves to a template parameter, then you cannot use an elaborated-type-specifier in this friend declaration; otherwise, the compiler issues an error.

Related reference:

"Friends" on page 373

Template arguments

There are three kinds of template arguments corresponding to the three types of template parameters:

- "Template type arguments"
- "Template non-type arguments" on page 438
- "Template template arguments" on page 440

A template argument must match the type and form specified by the corresponding parameter declared in the template.

C++11

When a parameter declared in a template is a template parameter pack, it corresponds to zero or more template arguments. For more information, see "Variadic templates (C++11)" on page 468

C++11

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 angle 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.

Related reference:

"Block/local scope" on page 2

"No linkage" on page 8

Bit field members

"typedef definitions" on page 74

Template type arguments

You cannot use one of the following types 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()`.

Related reference:

"Block/local scope" on page 2

"No linkage" on page 8

Bit field members

"typedef definitions" on page 74

Template non-type arguments

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.

Related reference:

"Integer constant expressions" on page 144

"References (C++ only)" on page 107

"External linkage" on page 8

"Static members" on page 366

Template template arguments

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>
#include <typeinfo>

using namespace std;

template<class T, class U> class A {
public:
   int x;
};

template<class U> class A<int, U> {
public:
   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.

Related reference:

"Partial specialization" on page 465

"Template instantiation" on page 456

Class templates

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`).

C++11

Template parameter packs can also be a kind of parameter for class templates. For more information, see "Variadic templates (C++11)" on page 468.

C++11

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.

C++11

By using template parameter packs, template declarations for a class template can have fewer or more arguments than the number of parameters specified in the

class template.

C++11

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 extraction 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
}
```

C++11

When the right angle bracket feature is enabled, the >> token is treated as two consecutive > tokens if both the following conditions are true:

- The >> token is in a context where one or more left angle brackets are active. A left angle bracket is active when it is not yet matched by a right angle bracket.
- The >> token is not nested within a delimited expression context.

If the first > token is in the context of a `template_parameter_list`, it is treated as the ending delimiter for the `template_parameter_list`. Otherwise, it is treated as the greater-than operator. The second > token terminates an enclosing `template_id` construct or a different construct, such as the `const_cast`, `dynamic_cast` `reinterpret_cast`, or `static_cast` operator. For example:

```
template<typename T> struct list {};
template<typename T>

struct vector
{
  operator T() const;
};

int main()
{
  // Valid, same as vector<vector<int> > v;
  vector<vector<int>> v;

  // Valid, treat the >> token as two consecutive > tokens.
  // The first > token is treated as the ending delimiter for the
  // template_parameter_list, and the second > token is treated as
  // the ending delimiter for the static_cast operator.
  const vector<int> vi = static_cast<vector<int>>(v);
}
```

A parenthesized expression is a delimited expression context. To use a bitwise shift operator inside `template-argument-list`, use parentheses to enclose the operator. For example:

```
template <int i> class X {};
template <class T> class Y {};

Y<X<(6>>1)>> y;          //Valid: 6>>1 uses the right shift operator
```

C++11

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();`

Related reference:

"Declaring class types" on page 347

"Scope of class names" on page 351

"Member functions" on page 359

"The static_cast operator (C++ only)" on page 177

"The dynamic_cast operator (C++ only)" on page 182

"The const_cast operator (C++ only)" on page 181

"The reinterpret_cast operator (C++ only)" on page 179

Class template declarations and definitions

A class template must be declared before any instantiation 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
{ /* ... */ };
```

In the z/OS implementation, the compiler checks the syntax of the entire template class definition when the template include files are being compiled if the TEMPINC compiler option is used, or during the original compiler pass if the NOTEMPINC compiler option is used. Any errors in the class definition are flagged. The compiler does not generate code or data until it requires a specialization. At that point it generates appropriate code and data for the specialization by using the argument list supplied.

Related reference:

"Class templates" on page 441

Static data members and templates

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>`.

Related reference:

"Static members" on page 366

Member functions of class templates

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)`.

C++11

You can use trailing return types for template member functions, including those that have the following kinds of return types:

- Return types depending on the types of the function arguments
- Complicated return types

For more information, see "Trailing return type (C++11)" on page 239.

C++11

Related reference:

"Member functions" on page 359

Friends and templates

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.

Related reference:

"Friends" on page 373

Function templates

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`.

C++11

Template parameter packs can be a kind of template parameter for function templates, and *function parameter packs* can be a kind of function parameter for function templates. For more information, see "Variadic templates (C++11)" on page 468.

You can use trailing return types for function templates, include those that have the following kinds of return types:

- Return types depending on the types of the function arguments
- Complicated return types

For more information, see "Trailing return type (C++11)" on page 239.

C++11

Template argument deduction

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*
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

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`.

C++11

If a template type parameter of a function template is a cv-unqualified rvalue reference, but the argument in the function call is an lvalue, the corresponding lvalue reference is used instead of the rvalue reference. However, if the template type parameter is a cv-qualified rvalue reference, and the argument in the function call is an lvalue, the template instantiation fails. For example:

```
template <class T> double func1(T&&);
template <class T> double func2(const T&&);

int var;

// The compiler calls func1<int&>(int&)
double a = func1(var);

// The compiler calls func1<int>(int&&)
double b = func1(1);
```

```
// error
double c = func2(var);

// The compiler calls func2<int>(const int&&)
double d = func2(1);
```

In this example, the template type parameter of the function template `func1` is a cv-unqualified rvalue reference, and the template type parameter of the function template `func2` is a cv-qualified rvalue reference. In the initialization of variable `a`, the template argument `var` is an lvalue, so the lvalue reference type `int&` is used in the instantiation of the function template `func1`. In the initialization of variable `b`, the template argument `1` is an rvalue, so the rvalue reference type `int&&` remains in the template instantiation. In the initialization of `c`, the template type parameter `T&&` is cv-qualified, but `var` is an lvalue, so `var` cannot be bound to the rvalue reference `T&&`.

C++11

Deducing non-type template arguments

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);
}
```

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.

C++11

Template argument deduction also applies to the variadic templates feature. For more information, see "Variadic templates (C++11)" on page 468.

C++11

Related reference:

"References (C++ only)" on page 107

"Lvalues and rvalues" on page 141

Overloading function templates

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'`.

Related reference:

"Overload resolution" on page 337

Partial ordering of function templates

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.

Related reference:

"Template specialization" on page 460

"new expressions (C++ only)" on page 185

Template instantiation

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 to handle a specific set of template arguments is called a *specialization*.

Template instantiation has two forms: explicit instantiation and implicit instantiation.

Related reference:

"Template specialization" on page 460

Explicit instantiation

You can explicitly tell the compiler when it should generate a definition from a template. This is called *explicit instantiation*. Explicit instantiation includes two forms: *explicit instantiation declaration* and *explicit instantiation definition*.

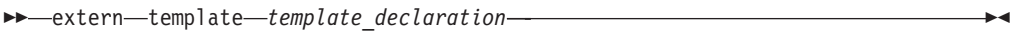
C++11

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

Explicit instantiation declaration

The explicit instantiation declarations feature is introduced in the C++11 standard. With this feature, you can suppress the implicit instantiation of a template specialization or its members. The `extern` keyword is used to indicate explicit instantiation declaration. The usage of `extern` here is different from that of a storage class specifier.

Explicit instantiation declaration syntax



You can provide an explicit instantiation declaration for a template specialization if an explicit instantiation definition of the template exists in other translation units or later in the same file. If one translation unit contains the explicit instantiation definition, other translation units can use the specialization without having the specialization instantiated multiple times. The following example demonstrates this concept:

```
//sample1.h:
template<typename T, T val>
union A{
   T func();
};
extern template union A<int, 55>;

template<class T, T val>
T A<T,val>::func(void){
   return val;
}

//sampleA.C"
#include "sample1.h"

template union A<int,55>;

//sampleB.C:
#include "sample1.h"

int main(void){
   return A<int, 55>().func();
}
```

sampleB.C uses the explicit instantiation definition of A<int, 55>().func()in sampleA.C.

If an explicit instantiation declaration of a function or class is declared, but there is no corresponding explicit instantiation definition anywhere in the program, the compiler issues an error message. See the following example:

```
// sample2.C
template <typename T, T val>
struct A{
   virtual T func();
   virtual T bar();
}

extern template int A<int,55>::func();

template <class T, T val>
T A<T,val>::func(void){
   return val;
}

template <class T, T val>
T A<T,val>::bar(void){
   return val;
}

int main(void){
   return A<int,55>().bar();
}
```

When you use explicit instantiation declaration, pay attention to the following restrictions:

- You can name a static class member in an explicit instantiation declaration, but you cannot name a static function because a static function cannot be accessed by name in other translation units.
- The explicit instantiation declaration of a class is not equivalent to the explicit instantiation declaration of each of its members.

C++11

Explicit instantiation definition

An explicit instantiation definition is an instantiation of a template specialization or its members.

Explicit instantiation definition syntax

Here is an example of explicit instantiation definition:

```
template<class T> class Array { void mf(); };
template class Array<char>;       /* explicit instantiation definition */
template void Array<int>::mf();   /* explicit instantiation definition */

template<class T> void sort(Array<T>& v) { }
template void sort(Array<char>&); /* explicit instantiation definition */

namespace N {
   template<class T> void f(T&) { }
}

template void N::f<int>(int&);
// The explicit instantiation definition is in namespace N.

int* p = 0;
template<class T> T g(T = &p);
template char g(char);           /* explicit instantiation definition */

template <class T> class X {
   private:
      T v(T arg) { return arg; };
};

template int X<int>::v(int);    /* explicit instantiation definition */

template<class T> T g(T val) { return val;}
template<class T> void Array<T>::mf() { }
```

An explicit instantiation definition of a template is in the same namespace where you define the template.

Access checking rules do not apply to the arguments in the explicit instantiation definitions. Template arguments in an explicit instantiation definition can be private types or objects. In this example, you can use the explicit instantiation definition `template int X<int>::v(int)` even though the member function is declared to be private.

The compiler does not use default arguments when you explicitly instantiate a template. In this example, you can use the explicit instantiation definition `template char g(char)` even though the default argument is an address of the type `int`.

Note: You cannot use the `inline` or C++11 `constexpr` C++11 specifier in an explicit instantiation of a function template or a member function of a class template.

C++11

Explicit instantiation and inline namespace definitions

Inline namespace definitions are namespace definitions with an initial `inline` keyword. Members of an inline namespace can be explicitly instantiated or specialized as if they were also members of the enclosing namespace. For more information, see "Inline namespace definitions (C++11)" on page 324.

C++11

Related reference:

"C++11 compatibility" on page 640

Implicit instantiation

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*.

C++11

The compiler does not need to generate the specialization for nonclass, noninline entities when an explicit instantiation declaration is present.

C++11

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();
1};

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.

Template specialization

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.

Related reference:

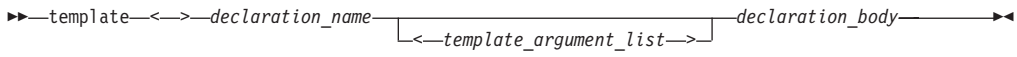
"Template instantiation" on page 456

Explicit specialization

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 templates:

- Function template
- 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

Explicit specialization declaration syntax



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;
}
```

See 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

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). See the following example:

```
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

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 code:

```
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

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

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 cases:

- 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

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');
}
```

See 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 of 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.

C++11

Explicit specialization and inline namespace definitions

Inline namespace definitions are namespace definitions with an initial `inline` keyword. Members of an inline namespace can be explicitly instantiated or specialized as if they were also members of the enclosing namespace. For more information, see “Inline namespace definitions (C++11)” on page 324.

C++11

Related reference:

“Function templates” on page 446

“Class templates” on page 441

“Member functions of class templates” on page 445

“Static data members and templates” on page 444

“Deleted functions” on page 222

Partial specialization

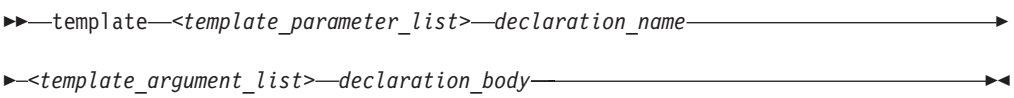
When you instantiate a class template, the compiler creates a definition based on the template arguments you have passed. Alternatively, if all those template arguments match those of an explicit specialization, the compiler uses the definition defined by the explicit specialization.

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.

Partial specialization syntax



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

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

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.

C++11

Partial specialization also applies to the variadic templates feature. For more information, see "Variadic templates (C++11)"

Related reference:

"Template parameters" on page 434

"Template arguments" on page 437

Variadic templates (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

Before C++11, templates had a fixed number of parameters that must be specified in the declaration of the templates. Templates could not directly express a class or function template that had a variable number of parameters. To partially alleviate this problem in the existing C++ programs, you could use overloaded function templates that had a different number of parameters or extra defaulted template parameters.

With the variadic templates feature, you can define class or function templates that have any number (including zero) of parameters. To achieve this goal, this feature introduces a kind of parameter called *parameter pack* to represent a list of zero or more parameters for templates.

The variadic template feature also introduces *pack expansion* to indicate that a parameter pack is expanded.

Two existing techniques, *template argument deduction* and *partial specialization*, can also apply to templates that have parameter packs in their parameter lists.

Parameter packs

A *parameter pack* can be a type of parameter for templates. Unlike previous parameters, which can only bind to a single argument, a parameter pack can pack multiple parameters into a single parameter by placing an ellipsis to the left of the parameter name.

In the template definition, a parameter pack is treated as a single parameter. In the template instantiation, a parameter pack is expanded and the correct number of the parameters are created.

According to the context where a parameter pack is used, the parameter pack can be either a *template parameter pack* or a *function parameter pack*.

Template parameter packs

A *template parameter pack* is a template parameter that represents any

number (including zero) of template parameters. Syntactically, a template parameter pack is a template parameter specified with an ellipsis. Consider the following example.

```
template<class...A> struct container{};
template<class...B> void func();
```

In this example, A and B are template parameter packs.

According to the type of the parameters contained in a template parameter pack, there are three kinds of template parameter packs:

- Type parameter packs
- Non-type parameter packs
- Template template parameter packs

A type parameter pack represents zero or more type template parameters. Similarly, a non-type parameter pack represents zero or more non-type template parameters.

Note: Template template parameter packs are not supported in z/OS XL C/C++ V2R1.

The following example shows a type parameter pack:

```
template<class...T> class X{};

X<> a;                    // the parameter list is empty
X<int> b;                 // the parameter list has one item
X<int, char, float> c;    // the parameter list has three items
```

In this example, the type parameter pack T is expanded into a list of zero or more type template parameters.

The following example shows a non-type parameter pack:

```
template<bool...A> class X{};

X<> a;                    // the parameter list is empty
X<true> b;                // the parameter list has one item
X<true, false, true> c;   // the parameter list has three items
```

In this example, the non-type parameter pack A is expanded into a list of zero or more non-type template parameters.

In a context where template arguments can be deduced; for example, function templates and class template partial specializations, a template parameter pack does not need to be the last template parameter of a template. In this case, you can declare more than one template parameter pack in the template parameter list. However, if template arguments cannot be deduced, you can declare at most one template parameter pack in the template parameter list, and the template parameter pack must be the last template parameter. Consider the following example:

```
// error
template<class...A, class...B>struct container1{};

// error
template<class...A,class B>struct container2{};
```

In this example, the compiler issues two error messages. One error message is for class template container1 because container1 has two template parameter packs A

and B that cannot be deduced. The other error message is for class template container2 because template parameter pack A is not the last template parameter of container2, and A cannot be deduced.

Default arguments cannot be used for a template parameter pack. Consider the following example:

```
template<typename...T=int> struct foo1{};
```

In this example, the compiler issues an error message because the template parameter pack T is given a default argument int.

Function parameter packs

A *function parameter pack* is a function parameter that represents zero or more function parameters. Syntactically, a function parameter pack is a function parameter specified with an ellipsis.

In the definition of a function template, a function parameter pack uses a template parameter pack in the function parameters. The template parameter pack is expanded by the function parameter pack. Consider the following example:

```
template<class...A> void func(A...args)
```

In this example, A is a template parameter pack, and args is a function parameter pack. You can call the function with any number (including zero) of arguments:

```
func();                 // void func();
func(1);                // void func(int);
func(1,2,3,4,5);        // void func(int,int,int,int,int);
func(1,'x', aWidget);   // void func(int,char,widget);
```

A function parameter pack is a *trailing function parameter pack* if it is the last function parameter of a function template. Otherwise, it is a *non-trailing function parameter pack*. A function template can have trailing and non-trailing function parameter packs. A non-trailing function parameter pack can be deduced only from the explicitly specified arguments when the function template is called. If the function template is called without explicit arguments, the non-trailing function parameter pack must be empty, as shown in the following example:

```
#include <cassert>

template<class...A, class...B> void func(A...arg1,int sz1, int sz2, B...arg2)
{
   assert( sizeof...(arg1) == sz1);
   assert( sizeof...(arg2) == sz2);
}

int main(void)
{
   //A:(int, int, int), B:(int, int, int, int, int)
   func<int,int,int>(1,2,3,3,5,1,2,3,4,5);

   //A: empty, B:(int, int, int, int, int)
   func(0,5,1,2,3,4,5);
   return 0;
}
```

In this example, function template func has two function parameter packs arg1 and arg2. arg1 is a non-trailing function parameter pack, and arg2 is a trailing function parameter pack. When func is called with three explicitly specified arguments as func<int,int,int>(1,2,3,3,5,1,2,3,4,5), both arg1 and arg2 are deduced successfully. When func is called without explicitly specified arguments as

`func(0,5,1,2,3,4,5)`, `arg2` is deduced successfully and `arg1` is empty. In this example, the template parameter packs of function template `func` can be deduced, so `func` can have more than one template parameter pack.

Pack expansion

A *pack expansion* is an expression that contains one or more parameter packs followed by an ellipsis to indicate that the parameter packs are expanded. Consider the following example:

```
template<class...T> void func(T...a){};
template<class...U> void func1(U...b){
    func(b...);
}
```

In this example, `T...` and `U...` are the corresponding pack expansions of the template parameter packs `T` and `U`, and `b...` is the pack expansion of the function parameter pack `b`.

A pack expansion can be used in the following contexts:

- Expression list
- Initializer list
- Base specifier list
- Member initializer list
- Template argument list
- Exception specification list

Expression list

Example:

```
#include <cstdio>
#include <cassert>

template<class...A> void func1(A...arg){
    assert(false);
}

void func1(int a1, int a2, int a3, int a4, int a5, int a6){
    printf("call with(%d,%d,%d,%d,%d,%d)\n",a1,a2,a3,a4,a5,a6);
}

template<class...A> int func(A...args){
    int size = sizeof...(A);
    switch(size){
        case 0: func1(99,99,99,99,99,99);
        break;
        case 1: func1(99,99,args...,99,99,99);
        break;
        case 2: func1(99,99,args...,99,99);
        break;
        case 3: func1(args...,99,99,99);
        break;
        case 4: func1(99,args...,99);
        break;
        case 5: func1(99,args...);
        break;
        case 6: func1(args...);
        break;
        default:
        func1(0,0,0,0,0,0);
    }
```

```
    return size;
}

int main(void){
    func();
    func(1);
    func(1,2);
    func(1,2,3);
    func(1,2,3,4);
    func(1,2,3,4,5);
    func(1,2,3,4,5,6);
    func(1,2,3,4,5,6,7);
    return 0;
}
```

The output of this example:

```
call with (99,99,99,99,99,99)
call with (99,99,1,99,99,99)
call with (99,99,1,2,99,99)
call with (1,2,3,99,99,99)
call with (99,1,2,3,4,99)
call with (99,1,2,3,4,5)
call with (1,2,3,4,5,6)
call with (0,0,0,0,0,0)
```

In this example, the switch statement shows the different positions of the pack expansion `args...` within the expression lists of the function `func1`. The output shows each call of the function `func1` to indicate the expansion.

Initializer list

Example:

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

void printarray(int arg[], int length){
    for(int n=0; n<length; n++){
        printf("%d ",arg[n]);
    }
    printf("\n");
}

template<class...A> void func(A...args){
    const int size = sizeof...(args) +5;
    printf("size %d\n", size);
    int res[sizeof...(args)+5]={99,98,args...,97,96,95};
    printarray(res,size);
}

int main(void)
{
    func();
    func(1);
    func(1,2);
    func(1,2,3);
    func(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20);
    return 0;
}
```

The output of this example:

```
size 5
99 98 97 96 95
size 6
99 98 1 97 96 95
```

```
size 7
99 98 1 2 97 96 95
size 8
99 98 1 2 3 97 96 95
size 25
99 98 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 97 96 95
```

In this example, the pack expansion `args...` is in the initializer list of the array `res`.

Base specifier list

Example:

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

struct a1{};
struct a2{};
struct a3{};
struct a4{};

template<class X> struct baseC{
    baseC() {printf("baseC primary ctor\n");}
};
template<> struct baseC<a1>{
    baseC() {printf("baseC a1 ctor\n");}
};
template<> struct baseC<a2>{
    baseC() {printf("baseC a2 ctor\n");}
};
template<> struct baseC<a3>{
    baseC() {printf("baseC a3 ctor\n");}
};
template<> struct baseC<a4>{
    baseC() {printf("baseC a4 ctor\n");}
};

template<class...A> struct container : public baseC<A>...{
    container(){
        printf("container ctor\n");
    }
};

int main(void){
    container<a1,a2,a3,a4> test;
    return 0;
}
```

The output of this example:

```
baseC a1 ctor
baseC a2 ctor
baseC a3 ctor
baseC a4 ctor
container ctor
```

In this example, the pack expansion `baseC<A>...` is in the base specifier list of the class template `container`. The pack expansion is expanded into four base classes `baseC<a1>`, `baseC<a2>`, `baseC<a3>`, and `baseC<a4>`. The output shows that all the four base class templates are initialized before the instantiation of the class template `container`.

Member initializer list

Example:

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

struct a1{};
struct a2{};
struct a3{};
struct a4{};

template<class X> struct baseC{
    baseC(int a) {printf("baseC primary ctor: %d\n", a);}
};
template<> struct baseC<a1>{
    baseC(int a) {printf("baseC a1 ctor: %d\n", a);}
};
template<> struct baseC<a2>{
    baseC(int a) {printf("baseC a2 ctor: %d\n", a);}
};
template<> struct baseC<a3>{
    baseC(int a) {printf("baseC a3 ctor: %d\n", a);}
};
template<> struct baseC<a4>{
    baseC(int a) {printf("baseC a4 ctor: %d\n", a);}
};

template<class...A> struct container : public baseC<A>...{
    container(): baseC<A>(12)...{
        printf("container ctor\n");
    }
};

int main(void){
    container<a1,a2,a3,a4> test;
    return 0;
}
```

The output of this example:

```
baseC a1 ctor:12
baseC a2 ctor:12
baseC a3 ctor:12
baseC a4 ctor:12
container ctor
```

In this example, the pack expansion `baseC<A>(12)...` is in the member initializer list of the class template `container`. The constructor initializer list is expanded to include the call for each base class `baseC<a1>(12)`, `baseC<a2>(12)`, `baseC<a3>(12)`, and `baseC<a4>(12)`.

Template argument list

Example:

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

template<int val> struct value{
    operator int(){return val;}
};

template <typename...I> struct container{
    container(){
        int array[sizeof...(I)]={I()...};
        printf("container<");
        for(int count = 0; count<sizeof...(I); count++){
            if(count>0){
```

```
                printf(",");
            }
            printf("%d", array[count]);
        }
        printf(">\n");
    }
};

template<class A, class B, class...C> void func(A arg1, B arg2, C...arg3){
    container<A,B,C...> t1;  // container<99,98,3,4,5,6>
    container<C...,A,B> t2;  // container<3,4,5,6,99,98>
    container<A,C...,B> t3;  // container<99,3,4,5,6,98>
}

int main(void){
    value<99> v99;
    value<98> v98;
    value<3> v3;
    value<4> v4;
    value<5> v5;
    value<6> v6;
    func(v99,v98,v3,v4,v5,v6);
    return 0;
}
```

The output of this example:

```
container<99,98,3,4,5,6>
container<3,4,5,6,99,98>
container<99,3,4,5,6,98>
```

In this example, the pack expansion `C...` is expanded in the context of template argument list for the class template `container`.

Exception specification list

Example:

```
struct a1{};
struct a2{};
struct a3{};
struct a4{};
struct a5{};
struct stuff{};

template<class...X> void func(int arg) throw(X...){
    a1 t1;
    a2 t2;
    a3 t3;
    a4 t4;
    a5 t5;
    stuff st;

    switch(arg){
        case 1:
            throw t1;
            break;
        case 2:
            throw t2;
            break;
        case 3:
            throw t3;
            break;
        case 4:
            throw t4;
            break;
        case 5:
```

```
            throw t5;
            break;
        default:
            throw st;
            break;
    }
}

int main(void){
    try{
        // if the throw specification is correctly expanded, none of
        // these calls should trigger an exception that is not expected
        func<a1,a2,a3,a4,a5,stuff>(1);
        func<a1,a2,a3,a4,a5,stuff>(2);
        func<a1,a2,a3,a4,a5,stuff>(3);
        func<a1,a2,a3,a4,a5,stuff>(4);
        func<a1,a2,a3,a4,a5,stuff>(5);
        func<a1,a2,a3,a4,a5,stuff>(99);
    }
    catch(...){
        return 0;
    }
    return 1;
}
```

In this example, the pack expansion `X...` is expanded in the context of exception specification list for the function template `func`.

If a parameter pack is declared, it must be expanded by a pack expansion. An appearance of a name of a parameter pack that is not expanded is incorrect. Consider the following example:

```
template<class...A> struct container;
template<class...B> struct container<B>{}
```

In this example, the compiler issues an error message because the template parameter pack `B` is not expanded.

Pack expansion cannot match a parameter that is not a parameter pack. Consider the following example:

```
template<class X> struct container{};

template<class A, class...B>
// Error, parameter A is not a parameter pack
void func1(container<A>...args){};

template<class A, class...B>
// Error, 1 is not a parameter pack
void func2(1...){};
```

If more than one parameter pack is referenced in a pack expansion, each expansion must have the same number of arguments expanded from these parameter packs. Consider the following example:

```
struct a1{}; struct a2{}; struct a3{}; struct a4{}; struct a5{};

template<class...X> struct baseC{};
template<class...A1> struct container{};
template<class...A, class...B, class...C>
struct container<baseC<A,B,C...>...>:public baseC<A,B...,C>{};

int main(void){
```

```
    container<baseC<a1,a4,a5,a5,a5>, baseC<a2,a3,a5,a5,a5>,
              baseC<a3,a2,a5,a5,a5>,baseC<a4,a1,a5,a5,a5> > test;
    return 0;
}
```

In this example, the template parameter packs `A`, `B`, and `C` are referenced in the same pack expansion `baseC<A,B,C...>...`. The compiler issues an error message to indicate that the lengths of these three template parameter packs are mismatched when expanding them during the template instantiation of the class template `container`.

Partial specialization

Partial specialization is a fundamental part of the variadic templates feature. A basic partial specialization can be used to access the individual arguments of a parameter pack. The following example shows how to use partial specialization for variadic templates:

```
// primary template
template<class...A> struct container;

// partial specialization
template<class B, class...C> struct container<B,C...>{};
```

When the class template `container` is instantiated with a list of arguments, the partial specialization is matched in all cases where there are one or more arguments. In that case, the template parameter `B` holds the first parameter, and the pack expansion `C...` contains the rest of the argument list. In the case of an empty list, the partial specialization is not matched, so the instantiation matches the primary template.

A pack expansion must be the last argument in the argument list for a partial specialization. Consider the following example:

```
template<class...A> struct container;

// partial specialization
template<class B, class...C> struct container<C...,B>{};
```

In this example, the compiler issues an error message because the pack expansion `C...` is not the last argument in the argument list for the partial specialization.

A partial specialization can have more than one template parameter pack in its parameter list. Consider the following example:

```
template<typename T1, typename T2> struct foo{};
template<typename...T> struct bar{};

// partial specialization
template<typename...T1,typename...T2> struct bar<foo<T1,T2>...>{};
```

In this example, the partial specialization has two template parameter packs `T1` and `T2` in its parameter list.

To access the arguments of a parameter pack, you can use partial specialization to access one member of the parameter pack in the first step, and recursively instantiate the remainder of the argument list to get all the elements, as shown in the following example:

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

```
struct a1{}; struct a2{}; struct a3{}; struct a4{}; struct a5{};
struct a6{}; struct a7{}; struct a8{}; struct a9{}; struct a10{};

template<typename X1, typename X2> struct foo{foo();};
template<typename X3, typename X4> foo<X3,X4>::foo(){cout<<"primary foo"<<endl;};
template<> struct foo<a1,a2>{foo(){cout<<"ctor foo<a1,a2>"<<endl;}};
template<> struct foo<a3,a4>{foo(){cout<<"ctor foo<a3,a4>"<<endl;}};
template<> struct foo<a5,a6>{foo(){cout<<"ctor foo<a5,a6>"<<endl;}};
template<> struct foo<a7,a8>{foo(){cout<<"ctor foo<a7,a8>"<<endl;}};
template<> struct foo<a9,a10>{foo(){cout<<"ctor foo<a9,a10>"<<endl;}};

template<typename...T>struct bar{bar{}{cout<<"bar primary"<<endl;}};

template<typename A, typename B, typename...T1, typename...T2>
struct bar<foo<A,B>,foo<T1,T2>...>{
    foo<A,B> data;
    bar<foo<T1,T2>...>data1;
};

template<> struct bar<foo<a9,a10> > {bar(){cout<<"ctor bar<foo<a9,a10>>"<<endl;}};

int main(){
    bar<foo<a1,a2>,foo<a3,a4>,foo<a5,a6>,foo<a7,a8>,foo<a9,a10> > t2;
    return 0;
}
```

The output of the example:

```
ctor foo<a1,a2>
ctor foo<a3,a4>
ctor foo<a5,a6>
ctor foo<a7,a8>
ctor bar<foo<a9,a10>
```

Template argument deduction

Parameter packs can be deduced by template argument deduction in the same way as other normal template parameters. The following example shows how template argument deduction expands packs from a function call:

```
template<class...A> void func(A...args){}

int main(void){
    func(1,2,3,4,5,6);
    return 0;
}
```

In this example, the function argument list is `(1,2,3,4,5,6)`. Each function argument is deduced to the type `int`, so the template parameter pack `A` is deduced to the following list of types: `(int,int,int,int,int,int)`. With all the expansions, the function template `func` is instantiated as `void func(int,int,int,int,int,int)`, which is the template function with the expanded function parameter pack.

In this example, if you change the function call statement `func(1,2,3,4,5,6)` to `func()`, template argument deduction deduces that the template parameter pack `A` is empty:

```
template<class...A> void func(A...args){}

int main(void){
    func();
    return 0;
}
```

Template argument deduction can expand packs from a template instantiation, as shown in the following example:

```
#include <cstdio>

template<int...A> struct container{
    void display(){printf("YIKES\n");}
};

template<int B, int...C> struct container<B,C...>{
    void display(){
        printf("spec %d\n",B);
        container<C...>test;
        test.display();
    }
 };

template<int C> struct container<C>{
    void display(){printf("spec %d\n",C);}
};

int main(void)
{
   printf("start\n\n");
   container<1,2,3,4,5,6,7,8,9,10> test;
   test.display();
   return 0;
}
```

The output of this example:

```
start

spec 1
spec 2
spec 3
spec 4
spec 5
spec 6
spec 7
spec 8
spec 9
spec 10
```

In this example, the partial specialization of the class template `container` is `template<int B, int...C> struct container<B,C...>`. The partial specialization is matched when the class template is instantiated to `container<1,2,3,4,5,6,7,8,9,10>`. Template argument deduction deduces the template parameter pack `C` and the parameter `B` from the argument list of the partial specialization. Template argument deduction then deduces the parameter `B` to be 1, the pack expansion `C...` to a list: `(2,3,4,5,6,7,8,9,10)`, and the template parameter pack `C` to the following list of types: `(int,int,int,int,int,int,int,int,int)`.

If you change the statement `container<1,2,3,4,5,6,7,8,9,10> test` to `container<1> test`, template argument deduction deduces that the template parameter pack `C` is empty.

Template argument deduction can expand packs after the explicit template arguments are found. Consider the following example:

```
#include <cassert>

template<class...A> int func(A...arg){
    return sizeof...(arg);
```

```
}

int main(void){
    assert(func<int>(1,2,3,4,5) == 5);
    return 0;
}
```

In this example, the template parameter pack `A` is deduced to a list of types: `(int,int,int,int,int)` using the explicit argument list and the arguments in the function call.

Related reference:

"The sizeof operator" on page 157

"Template parameters" on page 434

"Template arguments" on page 437

"Class templates" on page 441

"Function templates" on page 446

"Template argument deduction" on page 448

"Partial specialization" on page 465

"C++11 compatibility" on page 640

Name binding and dependent names

Name binding is the process of finding the declaration for each name that is explicitly or implicitly used in a template. The compiler might bind a name in the definition of a template, or it might 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. Consider the following example:

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

void f(double) { cout << "Function f(double)" << endl; }

template <class A> struct container{ // point of definition of container
   void member1(){
      // This call is not template dependent,
      // because it does not make any use of a template parameter.
      // The name is resolved at the point of definition, so f(int) is not visible.
      f(1);
   }
   void member2(A arg);
};
```

```
void f(int) { cout << "Function f(int)" << endl; }

void h(double) { cout << "Function h(double)" << endl; }

template <class A> void container<A>::member2(A arg){
   // This call is template dependent, so qualified name lookup only finds
   // names visible at the point of instantiation.
   ::h(arg);
}

template struct container<int>; // point of instantiation of container<int>

void h(int) { cout << "Function h(int)" << endl; }

int main(void){
   container<int> test;
   test.member1();
   test.member2(10);
   return 0;
}
```

The output of this example:

```
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 template `container` is located immediately before the keyword `template`. Because the function call `f(1)` does not depend on a template parameter, the compiler considers names declared before the definition of the template `container`. Therefore, the function call `f(1)` calls `f(double)`. Although `f(int)` is a better match, it is not in scope at the point of definition of `container`.

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 `container<int>` is the location of the explicit instantiation. Because the qualified function call `::h(arg)` depends on the template argument `arg`, the compiler considers names declared before the instantiation of `container<int>`. Therefore, the function call `h(arg)` calls `h(double)`. It does not consider `h(int)`, because this function is not in scope at the point of instantiation of `container<int>`.

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.

C++11

The decltype feature can interact with template dependent names. If the operand *expression* in the decltype(*expression*) type specifier is dependent on template parameters, the compiler cannot determine the validity of *expression* before the template instantiation, as shown in the following example:

```
template <class T, class U> int h(T t, U u, decltype(t+u) v);
```

In this example, the compiler issues an error message if the operand `t+u` is invalid after the instantiation of the function template `h`.

For more information, see "The decltype(*expression*) type specifier (C++11)" on page 78

C++11

Related reference:

"Template instantiation" on page 456

The typename keyword

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. Consider the following example:

```
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++ compiler interprets this statement as a function call. In order for the compiler to interpret this statement as a declaration, you must add the keyword `typename` to the beginning of `T:x(y)`. 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 the template parameter declarations.

Related reference:

"Template parameters" on page 434

The template keyword as qualifier

Use the keyword `template` as a qualifier to distinguish member templates from other entities. 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>();  // ill-formed
}
```

In this example, the definition of the variable `object_x` is ill-formed. The compiler assumes that the symbol `<` is a less-than operator. In order for the compiler to recognize the template function call, you must add the `template` qualifier:

```
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);
}
```

See the output of this 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 16. Exception handling (C++ only)

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:

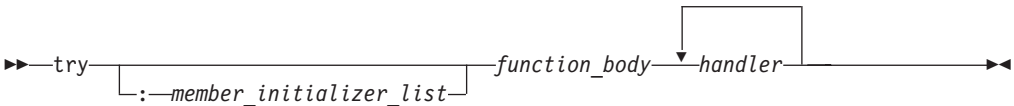
- `try` blocks
- `catch` blocks
- `throw` expressions
- "Exception specifications" on page 498

try blocks

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.

try block syntax

Function try block syntax



The following code 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(...) { }
}
```

See the following 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 reference:

"Initialization of base classes and members" on page 414

Nested try blocks

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();
```

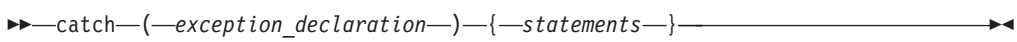
```
      try
      {
             func2();
      }
      catch (spec_err) { /* ... */ }
      func3();
}
catch (type_err) { /* ... */ }
// if no throw is issued, control resumes here.
```

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.

catch blocks

catch block syntax



You can declare a handler to catch many types of exceptions. The objects that a function can catch are declared in the parentheses following the `catch` keyword (the *exception_declaration*). You can catch both scalar and class objects. You can also catch cv-qualified objects. An exception declaration can declare an lvalue reference, in which case the exception object is passed by reference to the catch handler. The *exception_declaration* cannot be an incomplete type, abstract class type, C++11 rvalue reference type C++11 , or a reference or pointer to an incomplete type other than the following types:

- `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 reference:

"Type qualifiers" on page 85

"Member access" on page 371

"References (C++ only)" on page 107

Function try block handlers

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. This example is intended to show that the exception in `~B()` is caught by the function try block of `main()`, but that the exception in `~A()` is not caught because `~A()` is executed after `main()` has completed.

```
#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;
}
```

See 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;
}
```

See 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(...) { }
}
```

See 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;
}
```

See 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
```

C++11

If the delegating process exists and an exception occurs in the body of a target constructor, the exception can be caught by an appropriate handler in the try block of the delegating constructor. The following example demonstrates this:

```
#include <cstdio>
using std::printf;

int global_argc;

struct A{
   int _x;
   A();
   A(int);
};

A::A(int x):_x((printf("In A::A(int) initializer for A::_x.\n"),x)){
   printf("In A::A(int) constructor body.\n");

   if(global_argc % 2 !=0){
      printf("Will throw.\n");
      throw 0;
   }
   printf("Will not throw.\n");
}

A::A() try:A((printf("In A::A() initializer for delegating to A::A(int).\n"),42)){
   printf("In A::A() function-try-block body.\n");
}
catch(...){
   printf("In catch(...) handler for A::A() function-try-block.\n");
}

int main(int argc, char **argv){
   printf("In main().\n");
   global_argc = argc;
   try{
      A a;
      printf("Back in main().\n");
   }
   catch(...){
      printf("In catch(...) handler for try-block in main().\n");
   }
   return 0;
}
```

The example can produce different output depending on how many arguments are passed on the invocation of the resulting program. With an even number of arguments, the exception is thrown. The output is:

```
In main().
In A::A() initializer for delegating to A:A(int).
In A::A(int) initializer for A::_x.
```

```
In A::A(int) constructor body.
Will throw.
In catch(...) handler for A::A() function-try-block.
In catch(...) handler for try-block in main().
```

With an odd number of arguments, there is no exception thrown. The output is:

```
In main().
In A::A() initializer for delegating to A::A(int).
In A::A(int) initializer for A::_x.
In A::A(int) constructor body.
Will not throw.
In A::A() function-try-block body.
Back in main().
```

For more information, see "Delegating constructors (C++11)" on page 409

C++11

Related reference:

"The main() function" on page 247

"The static storage class specifier" on page 49

Chapter 9, "Namespaces (C++ only)," on page 317

"Destructors" on page 420

Arguments of catch blocks

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.

Matching between exceptions thrown and caught

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 reference:

"Pointer conversions" on page 137

"Type qualifiers" on page 85

"References (C++ only)" on page 107

Order of catching

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 reference:

"try blocks" on page 485

throw expressions

You use a *throw expression* to indicate that your program has encountered an exception.

throw expression syntax

The type of *assignment_expression* cannot be an incomplete type, abstract class type, or a pointer to an incomplete type other than the following types:

- `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. The constructor used to copy C++11 or move C++11 that object is chosen by overload resolution.

If the *assignment_expression* is an integral constant expression of integer type that evaluates to zero, this *assignment_expression* does not match a handler of pointer or pointer to member type.

Related reference:

Incomplete types

Rethrowing an exception

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`.

Stack unwinding

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 output of this 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()`.)

C++11

When the delegating constructors feature is enabled, if an exception is thrown in the body of a delegating constructor, the destructors of the objects constructed through target constructor will be invoked automatically. The destructors must be called in such a way that it calls the destructors of subobjects as appropriate. In particular, it should call the destructors for virtual base classes if the virtual base classes are created through the target constructor.

If an exception is thrown in the body of a delegating constructor, the destructor is invoked for the object created by the target constructor. If an exception escapes from a non-delegating constructor, the unwinding mechanism will call the destructors for the completely constructed subobjects. The following example demonstrates this:

```
class D{
  D():D('a') { printf("D:D().\n");}

  D:D(char) try: D(55){
    printf("D::D(char). Throws.\n");
    throw 0;
  }
  catch(...){
    printf("D::D(char).Catch block.\n");
  }

  D:D(int i):i(i_) {printf("D::D(int).\n");}

  D:~D() {printf("D::~D().\n");}
}

int main(void){
  D d;
}
```

The output of the example is:

```
D::D(int).
D::D(char).Throws.
D::~D().
D::D(char).Catch block.
```

In this example, an exception occurs in the delegating constructor `D:D(char)`, so destructor `D:~D()` is invoked for object `d`.

For more information, see “Delegating constructors (C++11)” on page 409

C++11

Exception specifications

C++ provides a mechanism to ensure that a given function is limited to throw 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`.

Exception specification syntax

The *type_id_list* is a comma-separated list of types. In this list you cannot specify an incomplete type, abstract class type, C++11 rvalue reference type C++11 , or a pointer or reference to an incomplete type other than the following types:

- `void*`
- `const void*`
- `volatile void*`
- `const volatile void*`

You can qualify the types in *type_id_list* with cv-qualifiers, but 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 the top-level function declarator in a declaration or definition of a function, pointer to function, reference to function, or pointer to member function. 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 reference:

Incomplete types

"Function declarations and definitions" on page 219

"Pointers to functions" on page 257

Chapter 14, "Special member functions (C++ only)," on page 407

"References (C++ only)" on page 107

Special exception handling functions

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:

- "The unexpected() function"
- "The terminate() function" on page 502

The unexpected() function

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.

Related reference:

"Special exception handling functions" on page 501

"The set_unexpected() and set_terminate() functions" on page 503

The terminate() function

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 cases are more specific:
  - 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:
- 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 following example demonstrates that if a function registered with atexit() throws an exception and the exception is not handled, an invocation to void terminate() is made.

```
extern "C" printf(char* ...);
#include <exception>
#include <cstdlib>
using namespace std;

extern "C" void f() {
  printf("Function f()\n");
  throw "Exception thrown from f()";
}

extern "C" void g() { printf("Function g()\n"); }
```

```
extern "C" 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");
}
```

See 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.

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.

Related reference:

"The set_unexpected() and set_terminate() functions"

The set_unexpected() and set_terminate() functions

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

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()`, and `bad_exception` has been specified, the exception thrown by `my_unexpected()` is replaced by an exception of type `bad_exception`.
4. 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. `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 neither class A nor `bad_exception` has been specified in the exception specification of function `g()`, `unexpected()` calls `terminate()`, which calls the function `my_terminate()`.
6. `my_terminate()` displays a message then calls `abort()`, which terminates the program.

Chapter 17. Preprocessor directives

Preprocessing is an initial phase to process text before compilation. Preprocessor directives are lines of the source file where the first non-whitespace character is #, which distinguishes them from other lines of text. The effect of each preprocessor directive is a change to the text and the result is a transformation of the text that does not contain the directives nor comments. The compiler can optionally output the preprocessed text to a file that has a `.i` suffix. Preprocessing is always the initial phase of compilation, even when the text has already been preprocessed.

Preprocessor directives consist of the following:

- "Macro definition directives," which replace tokens in the current file with specified replacement tokens
- "File inclusion directives" on page 516, which imbed files within the current file
- "Conditional compilation directives" on page 518, which conditionally compile sections of the current file
- "Message generation directives" on page 523, which control the generation of diagnostic messages
- "The null directive (#)" on page 525, which performs no action
- "Pragma directives" on page 525, which apply compiler-specific rules to specified sections of code
- "C99 preprocessor features adopted in C++11" on page 526

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 directives

Macro definition directives include the following directives and operators:

The #define directive

A *preprocessor define directive* directs the preprocessor to replace all subsequent occurrences of a macro with specified replacement tokens.

The `#define` directive can contain:

- "Object-like macros" on page 508

- "Function-like macros"

The following are some differences between using a macro for a constant and a declared constant:

- 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 representation used by the compiler because they are expanded inline. The inline expansion makes the macro value unavailable to the debugger.
- C A macro can be used in a compile-time constant expression, such as a bit field length, whereas a `const` object cannot.
- C++ The compiler does not type-check a macro, including macro arguments.

Object-like macros

An *object-like macro definition* replaces a single identifier with the specified replacement tokens. For example, 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];
```

is after this macro definition and in the same compilation unit, 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
doubl d = a.2
```

C++11 In C++11, the diagnostic for object-like macros in the C99 preprocessor is adopted to provide a common preprocessor interface for C and C++ compilers. The C++11 compiler issues a warning message if there are no white spaces between an object-like macro name and its replacement list in a macro definition. For more information, see "C99 preprocessor features adopted in C++11" on page 526.
C++11

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. C99 adds support for function-like macros with a variable number of arguments.

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 (...) as the formal parameter. 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 match or 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. */
```

If the parameter 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 intended 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 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 comments, character constants, or string constants for macro definitions, undefining a macro, or macro invocations.

The following example program contains two macro definitions and a macro invocation that refers to both of the defined macros:

CCNRAA8

```
/**This example illustrates #define directives.**/

void printf(const char*, ...);
#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 preprocessed, this program is replaced by code equivalent to the following:

CCNRAA9

```
void printf(const char*, ...);

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
```

IBM

Variadic macro extensions

Variadic macro extensions refer to two extensions to C99 and Standard C++ 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. The other extension provides a way to remove the dangling comma in a variadic macro when no variable arguments are specified. Both extensions have been implemented to facilitate porting programs developed with GNU C and C++.

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");`

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.

IBM

C++11 In C++11, the variadic macros feature and changes concerning empty macro arguments are adopted from the C99 preprocessor to provide a common preprocessor interface for C and C++ compilers. Variadic macros and empty macro arguments are supported in C++11. For more information, see "C99 preprocessor features adopted in C++11" on page 526.

Related reference:

"The const type qualifier" on page 88

"Operator precedence and associativity" on page 190

"Parenthesized expressions ( )" on page 145

The #undef directive

A *preprocessor undef directive* causes the preprocessor to end the scope of a preprocessor definition.

#undef directive syntax

If the identifier is not currently defined as a macro, #undef is ignored.

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.

The # 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.

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\""

Related reference:

"The null directive (#)" on page 525

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.

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`

Related reference:

"The #define directive" on page 507

Standard predefined macro names

The C compiler provides the following predefined macro names as specified in the ISO C language standard. C++ The z/OS XL C/C++ compiler supports these macros as an extension to be compatible with the C99 standard. C++ Except for __FILE__ and __LINE__, the value of the predefined macros remains constant throughout the translation unit. The predefined macro names typically start and finish with 2 underscore characters.

__DATE__

A character string literal containing the date when the source file was preprocessed.

The value of __DATE__ changes depending on when the input is preprocessed. 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 included files that are part of the source program are preprocessed. 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. If you set the language level to COMMONC, this macro is undefined. (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__ (C only)
The value of this C99 macro is 1, indicating that the C compiler is a hosted implementation. Note that this macro is only defined if __STDC__ is also defined.

__STDC_VERSION__ (C only)
The integer constant of type `long int`: 199409L for the C89 language level, 199901L for C99. Note that this macro is only defined if __STDC__ is also defined.

__TIME__
A character string literal containing the time when the source file was preprocessed.

The value of `__TIME__` changes as included files that are part of the source program are preprocessed. The time is in the form:

```
"hh:mm:ss"
```

where:

hh Represents the hour.

mm Represents the minutes.

ss Represents the seconds.

__cplusplus (C++ only)
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.

Related reference:

"The #line directive" on page 523

Object-like macros

File inclusion directives

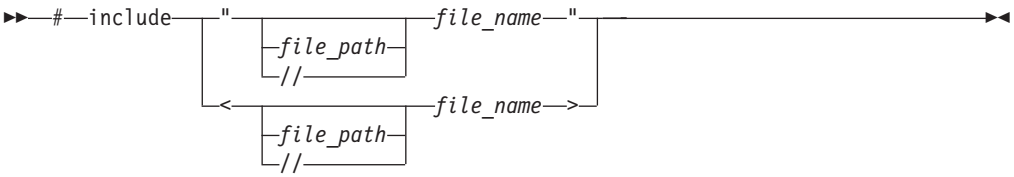
File inclusion directives consist of:

- "The #include directive," which inserts text from another source file
- IBM "The #include_next directive (IBM extension)" on page 517, which causes the compiler to omit the directory of the including file from the search path when searching for include files. IBM

The #include directive

A *preprocessor include directive* causes the preprocessor to replace the directive with the contents of the specified file.

#include directive syntax



You can specify a data set or a z/OS UNIX file for *file_name*. Use double slashes (//) before the *file_name* to indicate that the file is a data set. Use a single slash (/) anywhere in the *file_name* to indicate a z/OS UNIX file.

If the *file_name* is enclosed in double quotation marks, for example:

```
#include "payroll.h"
```

it is treated as a user-defined file, and may represent a header or source file.

If the *file_name* is enclosed in angle brackets, for example:

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

it is treated as a system-defined file, and must represent a header file.

The new-line and > characters cannot appear in a file name delimited by < and >. The new-line and " (double quotation marks) characters cannot appear in a file name delimited by " and ", although > can.

The *file_path* can be an absolute or relative path. If the double quotation marks are used, and *file_path* is a relative path, or is not specified, the preprocessor adds the directory of the including file to the list of paths to be searched for the included file. If the double angle brackets are used, and *file_path* is a relative path, or is not specified, the preprocessor does *not* add the directory of the including file to the list of paths to be searched for the included file.

The preprocessor resolves macros contained in an #include directive. After macro replacement, the resulting token sequence consists of a file name enclosed in either double quotation marks or the characters < and >. For example:

```
#define MONTH <july.h>
#include MONTH
```

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
extern int hour;
extern int min;
extern 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
```

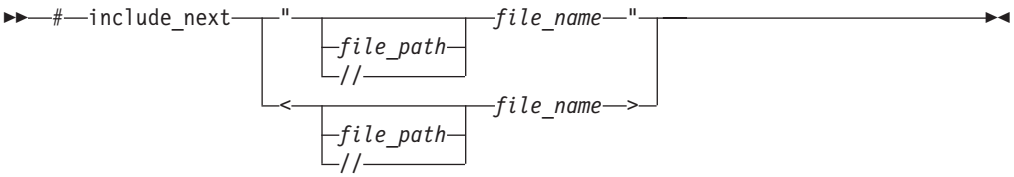
The z/OS implementation has specially defined behavior and compiler options for include file search paths, which are documented in greater detail in the descriptions of the SEARCH and LSEARCH options in the *z/OS XL C/C++ User's Guide*.

C++11 In C++11, the changes to header and include file names in the C99 preprocessor are adopted to provide a common preprocessor interface for C and C++ compilers. The first character of a header file name in an #include directive must not be a digit in C++11. For more information, see "C99 preprocessor features adopted in C++11" on page 526. C++11

The #include_next directive (IBM extension)

The preprocessor directive #include_next behaves like the #include directive, except that it specifically excludes the directory of the including file from the paths to be searched for the named file. All search paths up to and including the directory of the including file are omitted from the list of paths to be searched for the included file. This allows you to include multiple versions of a file with the same name in different parts of an application; or to include one header file in another header file with the same name (without the header including itself recursively). Provided that the different file versions are stored in different directories, the directive ensures you can access each version of the file, without requiring that you use absolute paths to specify the file name.

#include_next directive syntax



The directive must only be used in header files, and the file specified by the *file_name* must be a header file. There is no distinction between the use of double quotation marks and angle brackets to enclose the file name. You can specify a data set for *file_name*. Use double slashes (//) before the *file_name* to indicate that the file is a data set.

As an example of how search paths are resolved with the `#include_next` directive, assume that there are two versions of the file `t.h`: the first one, which is included in the source file `t.c`, is located in the subdirectory `path1`; the second one, which is included in the first one, is located in the subdirectory `path2`. Both directories are specified as include file search paths when `t.c` is compiled.

```
/* t.c  */

#include "t.h"

int main()
{
printf(", ret_val);
}

/* t.h in path1 */

#include_next "t.h"

int ret_val = RET;

/* t.h in path2 */

#define RET 55;
```

The `#include_next` directive instructs the preprocessor to skip the `path1` directory and start the search for the included file from the `path2` directory. This directive allows you to use two different versions of `t.h` and it prevents `t.h` from being included recursively.

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:

- "The #if and #elif directives" on page 519, which conditionally include or suppress portions of source code, depending on the result of a constant expression
- "The #ifdef directive" on page 520, which conditionally includes source text if a macro name is defined
- "The #ifndef directive" on page 521, which conditionally includes source text if a macro name is not defined

- "The #else directive" on page 521, which conditionally includes source text if the previous #if, #ifdef, #ifndef, or #elif test fails
- "The #endif directive" on page 521, which ends conditional text

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.

Conditional compilation directives can be nested. A #else, if present, can be matched unambiguously because of the required #endif.

```
#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.

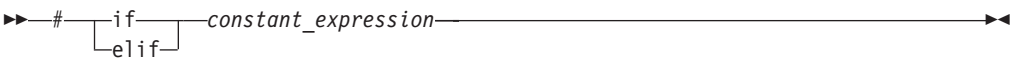
Only the first block whose expression is nonzero is processed. The remaining blocks at that nesting level are ignored. If none of the blocks at that nesting level has been processed and there is a #else directive, the block following the #else directive is processed. If none of the blocks at that nesting level has been processed and there is no #else directive, the entire nesting level is ignored.

The #if and #elif directives

The #if directive conditionally includes text for preprocessing. If the condition that follows the #if directive evaluates to a nonzero value, the text following up to but excluding the associated #endif is included as preprocessing input.

The #elif (a contraction of else-if) directive, if used, must be contained within a section of text subject to an #if directive. This directive optionally includes a section of text based on the evaluation result of the condition that immediately follows the directive. The #elif directive evaluates its condition only when the original condition on the #if evaluates to false and all conditions associated with preceding #elif directives subject to the original #if also evaluate to false.

#if and #elif directive syntax



All macros are expanded, except macros that are the operand of a defined operator. Any uses of the defined operator are processed, and all remaining keywords and identifiers are replaced with the token 0 C++ except true and false C++ .

The behavior is undefined if expanding the macros resulted in the token defined.

Notes:

- Casts cannot be performed. For example, the following code can be compiled successfully by both the C and C++ compilers.

  ```
  #if static_cast<int>(1)
  #error Unexpected
  #endif

  int main() {
  }
  ```

- Arithmetic is performed using long int type. C++11 In C++11, arithmetic is performed using long long int type. See "C99 preprocessor features adopted in C++11" on page 526 for detailed information. C++11
- The *constant_expression* can contain defined macros.
- 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)
  ```

- The *constant_expression* must be an integral constant expression.

If a macro is not defined, a value of 0 (zero) is assigned to it. In the following example, TEST is a macro identifier.

```
#include <stdio.h>
int main()
{
   #if TEST != 0    // No error even when TEST is not defined.
       printf("Macro TEST is defined to a non-zero value.");
   #endif
}
```

The #ifdef directive

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. You must use the `#endif` directive to end the conditional compilation directive.

#ifdef directive syntax

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
```

The #ifndef directive

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.

#ifndef directive syntax

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
```

The #else directive

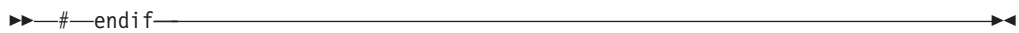
If the condition specified in the `#if`, `#ifdef`, or `#ifndef` directive evaluates to `0`, and the conditional compilation directive contains a preprocessor `#else` directive, the lines of code located between the preprocessor `#else` directive and the preprocessor `#endif` directive is selected by the preprocessor to be passed on to the compiler.

#else directive syntax

The #endif directive

The preprocessor `#endif` directive ends the conditional compilation directive.

#endif directive syntax



Extension of #endif and #else (IBM extension)

The C/C++ language standards do not allow extra text after `#endif` or `#else`. IBM XL C and XL C/C++ compilers comply with the standards. When you are porting code from a compiler that allows extra text after `#endif` or `#else`, you can specify option **LANGLVL(TEXTAFTERENDIF)** to suppress the warning message that is emitted.

One use is to comment on what is being tested by the corresponding `#if` or `#ifdef`. For example:

```
#ifdef MY_MACRO
...
#else MY_MACRO not defined
...
#endif MY_MACRO
```

In this case, if you want the compiler to be silent about this deviation from the standards, you can suppress the message by specifying option **LANGLVL(TEXTAFTERENDIF)**, while allowing the message to be emitted in other contexts, for example, if there is additional text after #undef.

The suboption **TEXTAFTERENDIF** can be specified with any of the supported language levels. In almost all cases the default for this suboption is **LANGLVL(NOTEXTAFTERENDIF)**, indicating that a message will be emitted if there is any extraneous text after `#else` or `#endif`. The one exception is in the C compiler, when the language level is "classic". In this case, the default for the suboption is **LANGLVL(TEXTAFTERENDIF)**, because this language level already allows extra text after `#else` or `#endif` without generating a message.

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:

CCNRABC

```
/**
 ** 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);
}
```

Message generation directives

Message generation directives include the following ones:

- "The #error directive," which defines text for a compile-time error message
- "The #line directive," which supplies a line number for compiler messages

Related reference:

"Conditional compilation directives" on page 518

The #error directive

A *preprocessor error directive* causes the preprocessor to generate an error message and causes the compilation to fail.

#error directive syntax

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 directives

```
#define BUFFER_SIZE 255

#if BUFFER_SIZE < 256
#error "BUFFER_SIZE is too small."
#endif
```

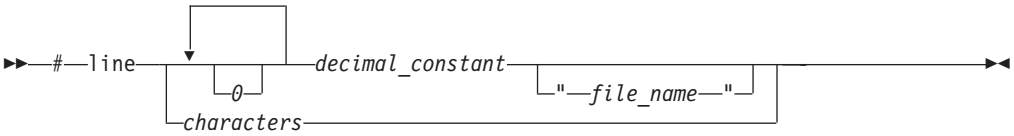
generate the error message:

```
BUFFER_SIZE is too small.
```

The #line directive

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.

#line directive syntax



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.

For z/OS XL C/C++ compilers, the *file_name* should be:

- A fully qualified sequential data set
- A fully qualified PDS or PDSE member
- A z/OS UNIX path name

The entire string is taken unchanged as the alternate source file name for the translation unit (for example, for use by the debugger). Consider if you are using it to redirect the debugger to source lines from this alternate file. In this case, you *must* ensure the file exists as specified and the line number on the `#line` directive matches the file contents. The compiler does not check this.

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.

You can use `#line` control directives to make the compiler provide more meaningful error messages. The following example program uses `#line` control directives to give each function an easily recognizable line number:

CCNRABD

```
/**
 ** 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
```

C++11 In C++11, the increased limit for #line directive from the C99 preprocessor are adopted to provide a common preprocessor interface for C and C++ compilers. The upper limit of `#line <integer>` preprocessor directives has been increased from 32,767 to 2,147,483,647 for the C++ preprocessor in conformance with the C99 preprocessor. For more information, see "C99 preprocessor features adopted in C++11" on page 526. C++11

The 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
```

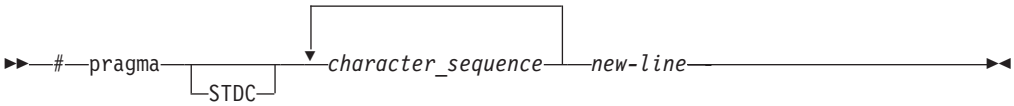
Related reference:

"The # operator" on page 513

Pragma directives

A *pragma* is an implementation-defined instruction to the compiler. It has the general form:

#pragma directive syntax



The *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 not subject to macro substitutions.

Note: C You can also use the `_Pragma` operator syntax to specify a pragma directive; for details, see "The _Pragma preprocessing operator" on page 526. C

More than one pragma construct can be specified on a single pragma directive. The compiler ignores unrecognized pragmas.

Standard C pragmas are described in "Standard pragmas." IBM Pragmas available for z/OS XL C/C++ are described in Chapter 18, "z/OS XL C/C++ pragmas," on page 531. IBM

The _Pragma preprocessing operator

The unary operator `_Pragma`, allows a preprocessor macro to be contained in a pragma directive.

_Pragma operator syntax

The *string_literal* can be prefixed with `L`, making it a wide-string literal.

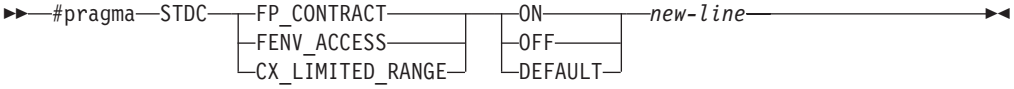
The string literal is destringized and tokenized. The resulting sequence of tokens is processed as if it appeared in a pragma directive. For example, the following two statements are equivalent:

```
_Pragma ( "pack(full)" )
#pragma pack(full)
```

C++11 In C++11, the *_Pragma operator* feature of the C99 preprocessor is adopted to provide a common preprocessor interface for C and C++ compilers. The `_Pragma` operator is an alternative method of specifying the `#pragma` directive. For more information, see "C99 preprocessor features adopted in C++11."

Standard pragmas

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:



These pragmas are recognized and ignored.

C99 preprocessor features adopted in C++11

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

In the C++11 standard, several C99 preprocessor features are adopted to provide a common preprocessor interface for C and C++ compilers. This eases porting C source files to the C++ compiler and eliminates some subtle semantic differences

that exist between the old C and C++ preprocessors, thus avoiding preprocessor compatibility issues or diverging preprocessor behaviors.

The following C99 preprocessor features are adopted in C++11:

- Preprocessor arithmetic with extended integer types
- Mixed string literal concatenation
- Diagnostic for header files and include names
- Increased limit for #line directives
- Diagnostic for object-like macro definitions
- The _Pragma operator
- Variadic macros and empty macro arguments
- Predefined macros

Preprocessor arithmetic with extended integer types

In the C89, C++98, and C++03 preprocessors, integer literals that have `int` or `unsigned int` type are widened to `long` or `unsigned long`. However, in the C99 and C++11 preprocessors, all `signed` and `unsigned` integer types (character types included) are widened to `long long` or `unsigned long long` under normal circumstances in XL C/C++.

If this feature is enabled, and both **-qnolonglong** and **-qlanglvl=noc99longlong** are set in either **-q32** or **-q64** modes, the preprocessor still uses `long long` or `unsigned long long` representations for all integral and character literals in preprocessor controlling expressions.

The following example is valid on the where the underlying type of `wchar_t` is unsigned short.

The following example shows a case where the `long long` support is enabled in **-q32** mode, this feature causes different inclusion branches to be chosen between the non-C++11 preprocessor and the C++11 preprocessor.

```
#if ~0ull == 0u + ~0u
#error C++11 preprocessor arithmetic! 0u has the same representation as 0ull,\
       hence ~0ull == 0u + ~0u
#else
#error non-C++11 preprocessor arithmetic. 0ul does not have the same \
       representation as 0ull, hence ~0ull != 0u + ~0u
#endif
```

If this feature is disabled and **-qwarn0x** is set, the C++11 preprocessor evaluates the controlling expressions in the `#if` and `#elif` directives, and compares the evaluation results against that of the non-C++11 preprocessor. If they are different, the compiler warns you that the preprocessor controlling expression evaluates differently between C++11 and non-C++11 language levels.

Mixed string literal concatenation

Regular strings can be concatenated with wide-string literals, for example:

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

int main()
{
wprintf(L"Guess what? %ls\n", "I can now concate" L"nate regular strings\
```

```
        and wide strings!");
printf("Guess what? %ls\n", L"I can now concate" "nate strings\
        this way too!");
}
```

This example prints the following output when it is executed:

```
Guess what? I can now concatenate regular strings and wide strings!
Guess what? I can now concatenate strings this way too!
```

Diagnostic for header files and include names

When this feature is enabled, if the first character of a header file name in an `#include` directive is a digit, the compiler issues a warning message. Consider the following example:

```
//inc.C

#include "0x/mylib.h"

int main()
{
return 0;
}
```

When compiling or preprocessing this example with this feature enabled, the compiler issues the following warning message:

```
"inc.C", line 1.10: 1540-0893 (W) The header file name "0x/mylib.h"
in #include directive shall not start with a digit.
```

Increased limit for #line directives

The upper limit of the `#line <integer>` preprocessor directives has been increased from 32,767 to 2,147,483,647 for the C++11 preprocessor in conformance with the C99 preprocessor.

```
#line 1000000   //Valid in C++11, but invalid in C++98
int main()
{
    return 0;
}
```

Diagnostic for object-like macro definitions

If there is no white space between object-like macro name and its replacement list in a macro definition, the C++11 compiler issues a warning message. Consider the following example:

```
//w.C

//With -qnodollar, '$' is not part of the macro name,
//thus it begins the replacement list
#define A$B c
#define STR2( x ) # x
#define STR( x ) STR2( x )
char x[] = STR( A$B );
```

When compiling or preprocessing this example with this feature enabled and **-qnodollar** is specified, the compiler issues the following warning message:

```
"w.C", line 1.10: 1540-0891 (W) Missing white space between
the identifier "A" and the replacement list.
```

The _Pragma operator

The _Pragma operator is an alternative method of specifying #pragma directives. For example, the following two statements are equivalent:

```
#pragma comment(copyright, "IBM 2010")
_Pragma("comment(copyright, \"IBM 2010\")")
```

The string IBM 2010 is inserted into the C++ object file when the following code is compiled:

```
_Pragma("comment(copyright, \"IBM 2010\")")
int main()
{
   return 0;
}
```

Variadic macros and empty macro arguments

Variadic macros and empty macro arguments are supported in C99 and C++11. For details, see Variadic macros.

Predefined macros

The __STDC_HOSTED__ macro is predefined to 1, regardless of whether the following macros are defined or not:

- __STDC__
- __STDC_VERSION__
- __STDC_ISO_10646_

Related reference:

"Integer literals" on page 19

"String literals" on page 28

"The #include directive" on page 516

"The #line directive" on page 523

"The #define directive" on page 507

"The _Pragma preprocessing operator" on page 526

"C++11 compatibility" on page 640

Chapter 18. z/OS XL C/C++ pragmas

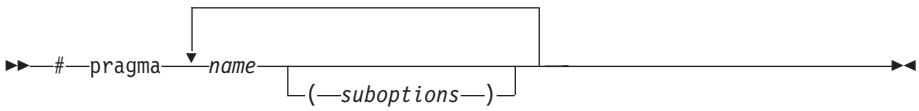
The following sections describe the pragmas available in z/OS XL C/C++:

Pragma directive syntax

z/OS XL C/C++ supports the following pragma directive:

#pragma *name*

This form uses the following syntax:



The *name* is the pragma directive name, and the *suboptions* are any required or optional suboptions that can be specified for the pragma, where applicable.

You can specify more than one *name* and *suboptions* in a single **#pragma** statement.

The compiler ignores unrecognized pragmas, issuing an informational message indicating this.

If you have any pragmas that are not common to both C and C++ in code that will be compiled by both compilers, you may add conditional compilation directives around the pragmas. (This is not strictly necessary since unrecognized pragmas are ignored.) For example, **#pragma object_model** is only recognized by the C++ compiler, so you may decide to add conditional compilation directives around the pragma.

```
#ifdef __cplusplus
#pragma object_model(pop)
#endif
```

Scope of pragma directives

Many pragma directives can be specified at any point within the source code in a compilation unit; others must be specified before any other directives or source code statements. In the individual descriptions for each pragma, the "Usage" section describes any constraints on the pragma's placement.

In general, if you specify a pragma directive before any code in your source program, it applies to the entire compilation unit, including any header files that are included. For a directive that can appear anywhere in your source code, it applies from the point at which it is specified, until the end of the compilation unit.

You can further restrict the scope of a pragma's application by using complementary pairs of pragma directives around a selected section of code. For example, using **#pragma checkout (suspend)** and **#pragma checkout (resume)** directives as follows requests that the selected parts of your source code be excluded from being diagnosed by the CHECKOUT compiler option:

```
#pragma checkout (suspend)

/*Source code between the suspend and resume pragma
  checkout is excluded from CHECKOUT analysis*/

#pragma checkout (resume)
```

Many pragmas provide "pop" or "reset" suboptions that allow you to enable and disable pragma settings in a stack-based fashion; examples of these are provided in the relevant pragma descriptions.

IPA effects

Interprocedural Analysis (IPA), through the IPA compiler option, is a mechanism for performing optimizations across the translation units of your C or C++ program. IPA also performs optimizations not otherwise available with the z/OS XL C/C++ compiler.

You may see changes during the IPA link step, due to the effect of a pragma. The IPA link step detects and resolves the conflicting effects of pragmas, and the conflicting effects of pragmas and compiler options that you specified for different translation units. There may also be conflicting effects between pragmas and equivalent compiler options that you specified for the IPA link step.

IPA resolves these conflicts similar to the way it resolves conflicting effects of compiler options that are specified for the IPA compile step and the IPA link step. The compiler Options Map section of the IPA link step listing shows the conflicting effects between compiler options and pragmas, along with the resolutions.

Summary of compiler pragmas by functional category

The z/OS XL C/C++ pragmas available on the z/OS platform are grouped into the following categories:

- "Language element control"
- "C++ template pragmas" on page 533
- "Floating point and integer control" on page 533
- "Error checking and debugging" on page 534
- "Listings, messages and compiler information" on page 534
- "Optimization and tuning" on page 534
- "Object code control" on page 535
- "Portability and migration" on page 536

For descriptions of these categories, see "Summary of compiler options" in the *z/OS XL C/C++ User's Guide.*

Language element control

Table 35. Language element control pragmas

Pragma	Description
"#pragma extension" on page 554	Enables extended language features.

Table 35. Language element control pragmas (continued)

Pragma	Description
"#pragma filetag" on page 556	Specifies the code set in which the source code was entered.
"#pragma langlvl (C only)" on page 564	Determines whether source code and compiler options should be checked for conformance to a specific language standard, or subset or superset of a standard.
"#pragma margins/nomargins" on page 572	Specifies the columns in the input line to scan for input to the compiler.
"#pragma options (C only)" on page 581	Specifies a list of compiler options that are to be processed as if you had typed them on the command line or on the CPARM parameter of the IBM-supplied catalogued procedures.
"#pragma runopts" on page 593	Specifies a list of runtime options for the compiler to use at execution time.
"#pragma sequence" on page 594	Defines the section of the input record that is to contain sequence numbers.
"#pragma XOPTS" on page 604	Passes suboptions directly to the CICS integrated translator for processing CICS statements embedded in C/C++ source code.

C++ template pragmas

Table 36. C++ template pragmas

Pragma	Description
"#pragma define (C++ only)" on page 546	Provides an alternative method for explicitly instantiating a template class.
"#pragma do_not_instantiate (C++ only)" on page 549	Prevents the specified template declaration from being instantiated.
"#pragma implementation (C++ only)" on page 558	For use with the TEMPINC compiler option, supplies the name of the file containing the template definitions corresponding to the template declarations contained in a header file.

Floating point and integer control

Table 37. Floating point and integer control pragmas

Pragma	Description
#pragma chars	Determines whether all variables of type `char` are treated as either `signed` or `unsigned`.
"#pragma enum" on page 549	Specifies the amount of storage occupied by enumerations.

Error checking and debugging

Table 38. Error checking and debugging pragmas

Pragma	Description
"#pragma operator_new (C++ only)" on page 578	Determines whether the `new` and `new[]` operators throw an exception if the requested memory cannot be allocated.

Listings, messages and compiler information

Table 39. Listings, messages and compiler information pragmas

Pragma	Description
"#pragma checkout" on page 540	Controls the diagnostic messages that are generated by the compiler.
"#pragma info (C++ only)" on page 558	Controls the diagnostic messages that are generated by the compiler.
"#pragma page (C only)" on page 587	Specifies that the code following the pragma begins at the top of the page in the generated source listing.
"#pragma pagesize (C only)" on page 587	Sets the number of lines per page for the generated source listing.
"#pragma report (C++ only)" on page 591	Controls the generation of diagnostic messages.
"#pragma skip (C only)" on page 595	Skips lines of the generated source listing.
"#pragma subtitle (C only)" on page 597	Places subtitle text on all subsequent pages of the generated source listing.
"#pragma title (C only)" on page 598	Places title text on all subsequent pages of the generated source listing.

Optimization and tuning

Table 40. Optimization and tuning pragmas

Pragma	Description
"#pragma disjoint" on page 547	Lists identifiers that are not aliased to each other within the scope of their use.
"#pragma execution_frequency" on page 552	Marks program source code that you expect will be either very frequently or very infrequently executed.
"#pragma inline (C only) / noinline" on page 559	Specifies that a C function is to be inlined, or that a C or C++ function is not to be inlined.
#pragma isolated_call	Specifies functions in the source file that have no side effects other than those implied by their parameters.
"#pragma leaves" on page 565	Informs the compiler that a named function never returns to the instruction following a call to that function.
"#pragma option_override" on page 579	Allows you to specify optimization options at the subprogram level that override optimization options given on the command line.
"#pragma reachable" on page 590	Informs the compiler that the point in the program after a named function can be the target of a branch from some unknown location.

Table 40. Optimization and tuning pragmas (continued)

Pragma	Description
#pragma unroll	Controls loop unrolling, for improved performance.

Object code control

Table 41. Object code control pragmas

Pragma	Description
"#pragma arch_section (IBM extension)" on page 537	Overrides the architecture of a statement in the source file. This pragma directive enables your program to leverage specific hardware features like using newer and faster instructions through built-ins or compiler generated instructions based on the architecture sections within the source file. You do not need to maintain code customized for a specific hardware model in separate source files. This pragma directive reduces the overhead of the call to an external routine and permits code for different machine architectures to coexist in one source file.
"#pragma comment" on page 541	Places a comment into the object module.
"#pragma csect" on page 545	Identifies the name for the code, static, or test control section (CSECT) of the object module.
"#pragma environment (C only)" on page 551	Uses C code as an assembler substitute.
"#pragma export" on page 553	Declares that an external function or variable is to be exported.
"#pragma hashome (C++ only)" on page 556	Informs the compiler that the specified class has a home module that will be specified by **#pragma ishome**.
"#pragma insert_asm (C only)" on page 560	Enables users to provide additional assembler statements that are inserted into the compiler-generated High-Level Assembler (HLASM) code.
"#pragma ishome (C++ only)" on page 561	Informs the compiler that the specified class's home module is the current compilation unit.
"#pragma linkage (C only)" on page 566	Identifies the entry point of modules that are used in interlanguage calls from C programs as well as the linkage or calling convention that is used on a function call.
"#pragma longname/nolongname" on page 569	Specifies whether the compiler is to generate mixed-case names that can be longer than 8 characters in the object module.
"#pragma map" on page 570	Converts all references to an identifier to another, externally defined identifier.
"#pragma pack" on page 583	Sets the alignment of all aggregate members to a specified byte boundary.
#pragma priority (C++ only)	Specifies the priority level for the initialization of static objects.
"#pragma prolog (C only), #pragma epilog (C only)" on page 589	When used with the METAL option, inserts High-Level Assembly (HLASM) prolog or epilog code for a specified function.
#pragma strings	Specifies the storage type for string literals.
"#pragma target (C only)" on page 597	Specifies the operating system or runtime environment for which the compiler creates the object module.

Table 41. Object code control pragmas (continued)

Pragma	Description
"#pragma variable" on page 601	Specifies whether the compiler is to use a named external object in a reentrant or non-reentrant fashion.

Portability and migration

Table 42. Portability and migration pragmas

Pragma	Description
"#pragma convert" on page 543	Provides a way to specify more than one coded character set in a single compilation unit to convert string literals.
"#pragma convlit" on page 544	Suspends the string literal conversion that the CONVLIT compiler option performs during specific portions of your program.
"#pragma namemangling (C++ only)" on page 573	Chooses the name mangling scheme for external symbol names generated from C++ source code.
"#pragma namemanglingrule (C++ only)" on page 575	Provides fined-grained control over the name mangling scheme in effect for selected portions of source code, specifically with respect to the mangling of cv-qualifiers in function parameters.
"#pragma object_model (C++ only)" on page 577	Sets the object model to be used for structures, unions, and classes.
"#pragma wsizeof" on page 602	Toggles the behavior of the `sizeof` operator between that of a compiler prior the C/C++ for MVS/ESA V3R2 and the z/OS XL C/C++ compiler.

Individual pragma descriptions

This section contains descriptions of individual pragmas available in z/OS XL C/C++.

For each pragma, the following information is given:

Category
The functional category to which the pragma belongs is listed here.

Purpose
This section provides a brief description of the effect of the pragma, and why you might want to use it.

Syntax
This section provides the syntax for the pragma. For convenience, the **#pragma** ***name*** form of the directive is used in each case. However (in C), it is perfectly valid to use the alternate C99-style `_Pragma` operator syntax; see "Pragma directive syntax" on page 531 for details.

Parameters
This section describes the suboptions that are available for the pragma, where applicable.

Usage This section describes any rules or usage considerations you should be aware of when using the pragma. These can include restrictions on the pragma's applicability, valid placement of the pragma, and so on.

IPA effects
For those pragmas where there are special considerations for IPA, the pragma descriptions include IPA-related information.

Examples
Where appropriate, examples of pragma directive use are provided in this section.

#pragma arch_section (IBM extension)

Category

"Object code control" on page 535

Purpose

Overrides the architecture of a statement in the source file. This pragma directive enables your program to leverage specific hardware features like using newer and faster instructions through built-ins or compiler generated instructions based on the architecture sections within the source file. You do not need to maintain code customized for a specific hardware model in separate source files. This pragma directive reduces the overhead of the call to an external routine and permits code for different machine architectures to coexist in one source file.

Syntax

Defaults

Not applicable.

Parameters

architecture level
Specifies the hardware architecture level for the code section that follows the directive. Valid architecture levels are 5 and above, which are described under the ARCHITECTURE option in *z/OS XL C/C++ User's Guide*.

statement
A single statement or a compound statement that is enclosed by curly braces.

Usage

Upon encountering the #pragma arch_section directive in the source, the compiler replaces the architecture level specified by compiler option ARCH or a parent scope #pragma options(architecture(n)) until the end of the statement.

The #pragma arch_section directive can be nested. When nesting, the outer sections should be lower than the inner sections, as the older hardware features are a subset of the newer ones.

The TUNE compiler option should be equal or greater than the highest architecture level specified in the #pragma arch_section directive.

Options that enable language extensions associated with a hardware feature should be specified on compiler command line invocation. For example, DFP to enable _Decimal{32, 64, 128} type, and VECTOR to enable vector type.

Specifying the following options on the compiler command line invocation does not enable the hardware features associated with them for the statements following the #pragma arch_section directive. These options and their associated hardware features are as follow:

- RTCHECK for #pragma arch_section 8 or higher
- HGPR for #pragma arch_section 5 or higer
- FLOAT(HEX,MAF),OPTIMIZE for #pragma arch_section 9 or higher

When using #pragma arch_section in the source, check the hardware model to ensure that the specified functionality is available at execution time. You can use one of the compiler supported built-ins functions, which are documented in *z/OS XL C/C++ Programming Guide*.

Examples

The following example shows that architecture levels must be specified in an increasing order. Suppose that you have the following program with name `nest.c`:

```
#include <builtins.h>

extern unsigned int c, a, b, d;
int countzero();

int main(void){
  unsigned long zero=0ul;
  __builtin_cpu_init();
  if (__builtin_cpu_supports("popcount"))
    #pragma arch_section(11)
    {
      c = a + b;
    #pragma arch_section(9)
      d = __popcnt(zero);
    }
  return 55;
}
```

If you compile this program with the following command:

```
xlc -c -qlanglvl=extended nest.c
```

The following warning message will be issued:

```
The value "9" specified on pragma "arch_section" is lower than the effective
architecture value "11". The pragma is ignored.
```

The following example shows that you cannot use decimal floating-point in sections of the source when effective architecture level is lower than 7. Suppose that you have the following program with name `dfp.c`:

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

_Decimal64 dec_n=-1.0;

int main(){
  __builtin_cpu_init();
  if(__builtin_cpu_supports("dfp"))
    #pragma arch_section(7){
      dec_n=-6.4;
```

```
      dec_n = __d64_abs(dec_n);
      printf("%f\n", dec_n);
    }
  return 55;
}
```

If you compile this program with the following command:

```
xlC -c dfp.c -qphsinfo -qarch=5 -qDFP
```

The following error message will be issued:

```
The use of decimal floating-point is invalid for architecture level "5".
```

Related information

- ARCHITECTURE option in the *z/OS XL C/C++ User's Guide*.
- Hardware model and feature built-ins section in *z/OS XL C/C++ Programming Guide*.

#pragma chars

Category

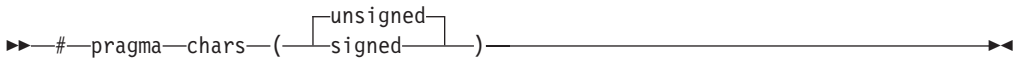
Floating-point and integer control

Purpose

Determines whether all variables of type `char` are treated as either `signed` or `unsigned`.

Syntax

Pragma syntax



Defaults

See the CHARS option in the *z/OS XL C/C++ User's Guide*.

Parameters

unsigned
Variables of type `char` are treated as `unsigned char`.

signed
Variables of type `char` are treated as `signed char`.

Usage

Regardless of the setting of this pragma, the type of `char` is still considered to be distinct from the types `unsigned char` and `signed char` for purposes of type-compatibility checking or C++ overloading.

If the pragma is specified more than once in the source file, the first one will take precedence. Once specified, the pragma applies to the entire file and cannot be disabled; if a source file contains any functions that you want to compile without

#pragma chars, place these functions in a different file. C The pragma must appear before any source statements, except for the pragmas **filetag**, **longname**, **langlvl** or **target**, which may precede it. C++ The pragma must appear before any source statements.

Related information

- The CHARS option in the *z/OS XL C/C++ User's Guide*.

#pragma checkout

Category

"Listings, messages and compiler information" on page 534

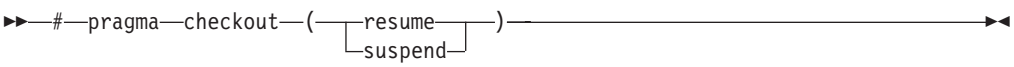
Purpose

Controls the diagnostic messages that are generated by the compiler.

You can suspend the diagnostics that the INFO or CHECKOUT compiler options perform during specific portions of your program. You can then resume the same level of diagnostics later in the file. C++ You can use this pragma directive in place of the INFO option or **#pragma info** directive. C++

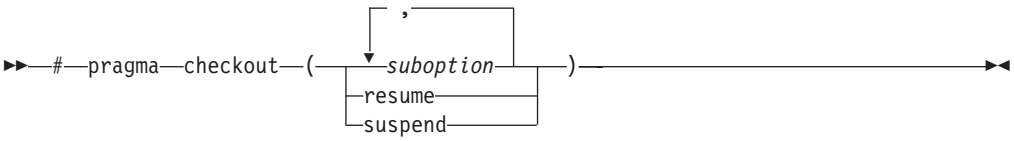
Syntax

C



C

C++



C++

Defaults

See the INFO and CHECKOUT options in the *z/OS XL C/C++ User's Guide.*

Parameters

C++ *suboption*
Any suboption supported by the INFO compiler option. For details, see the INFO option in the *z/OS XL C/C++ User's Guide.* C++

suspend
Instructs the compiler to suspend all diagnostic operations for the code following the directive.

resume
Instructs the compiler to resume all diagnostic operations for the code following the directive.

Usage

This pragma can appear anywhere that a preprocessor directive is valid.

Related information

- "#pragma info (C++ only)" on page 558
- The INFO and CHECKOUT options in the *z/OS XL C/C++ User's Guide.*

#pragma comment

Category

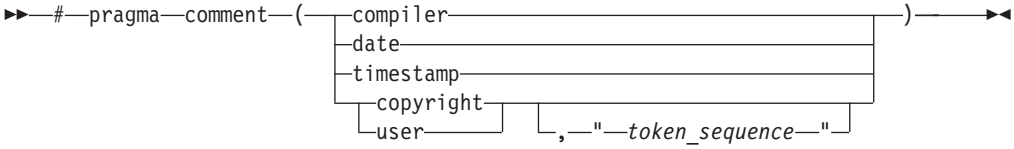
Object code control

Purpose

Places a comment into the object module.

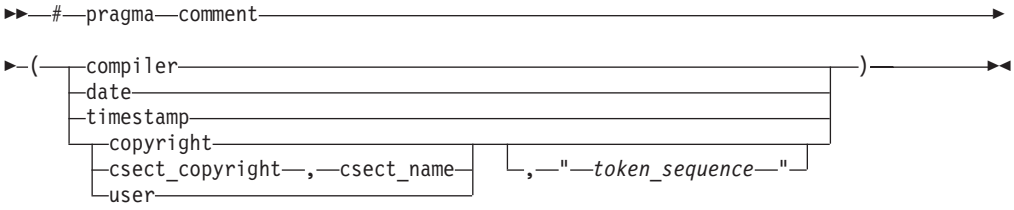
Syntax

C++



C++

C



C

Parameters

compiler
Appends the name and version of the compiler in an END information record at the end of the generated object module. The name and version are not included in the generated executable, nor are they loaded into memory when the program is run.

date
The date and time of the compilation are appended in an END information

record at the end of the generated object module. The date and time are not included in the generated executable, nor are they loaded into memory when the program is run.

timestamp
Appends the date and time of the last modification of the source in an END information record at the end of the generated object module. The date and time are not included in the generated executable, nor are they loaded into memory when the program is run.

If the compiler cannot find the timestamp for a source file, the directive returns `Mon Jan 1 0:00:01 1990`.

copyright
Places the text specified by the *token_sequence*, if any, into the generated object module. The *token_sequence* is included in the generated executable and loaded into memory when the program is run.

C **csect_copyright**
Places the text specified by the *token_sequence*, if any, into a CSECT section named *csect_name* in the generated object module. *token_sequence* is included in the generated executable and loaded into memory when the program is run.
C

C **csect_name**
The user-specified *csect_name* should not conflict with CODE, STATIC or TEST CSECT names specified using **#pragma csect**. C

user
Places the text specified by the *token_sequence*, if any, into the generated object module. The characters are placed in two locations in the generated object module. One copy of the string is placed in the code image so that the string will be included in the executable load module. This copy is not necessarily loaded into memory when the program is run. A second copy of the string is placed on the END records in columns 34 to 71 for XOBJ-format object modules, or in columns 4 to 80 for GOFF-format object modules.

token_sequence
The characters in this field, if specified, must be enclosed in double quotation marks ("). This field has a 32767-byte limit.

Usage

More than one **comment** directive can appear in a translation unit, and each type of **comment** directive can appear more than once, with the exception of **copyright**, which can appear only once.

You can display the object-file comments by using the MAP option for the C370LIB utility.

IPA effects

This directive affects the IPA compile step only if the OBJECT suboption of the IPA compiler option is in effect.

During the IPA link step, the compiler may combine multiple objects into one. In the case where multiple **csect_copyright** comment directives are found, the compiler will keep one and discard the rest.

During the partitioning process in the IPA link step, the compiler places the text string information `#pragma comment` at the beginning of a partition.

#pragma convert

Category

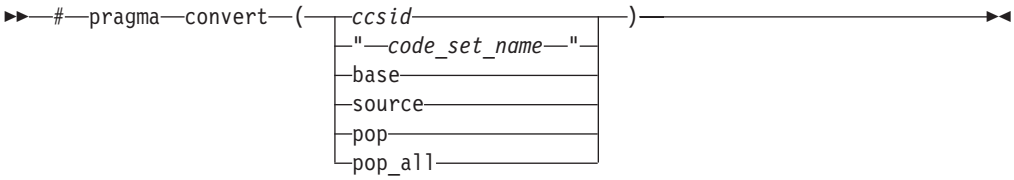
"Portability and migration" on page 536

Purpose

Provides a way to specify more than one coded character set in a single compilation unit to convert string literals.

Unlike the related CONVLIT, ASCII/EBDCIC, and LOCALE compiler options, it allows for more than one character encoding to be used for string literals in the same compilation unit.

Syntax



Defaults

See the *z/OS XL C/C++ User's Guide* for information about default code pages.

Parameters

ccsid
Represents the Coded Character Set Identifier, which is an integer value between 0 and 65535 inclusive. The coded character set can be based on either EBCDIC or ASCII.

code_set_name
Is a string that specifies an ASCII or EBCDIC based codepage.

base
Represents the codepage determined by the current locale or the LOCALE compiler option. If the ASCII option has been specified, then **base** is the ISO8859-1 codepage; if the CONVLIT option has been specified, then **base** is the codepage indicated by that option. If both ASCII and CONVLIT options have been specified, then **base** is the codepage indicated by the CONVLIT option.

source
Represents the codepage the source file is written in; that is, the filetag. If there is no filetag, then **source** is the codepage indicated by the LOCALE option specified at compile time.

pop
Resets the code set to that which was previously in effect immediately before the current codepage.

pop_all
Resets the codepage to that which was in effect before the introduction of any **convert** pragmas.

Usage

The compiler options CONVLIT, ASCII/EBCDIC, and LOCALE determine the code set in effect before any **#pragma convert** directives are introduced, and after all **#pragma convert** directives are popped from the stack.

The conversion continues from the point of placement of the first **#pragma convert** directive until another **#pragma convert** directive is encountered, or until the end of the source file is reached. For every **#pragma convert** directive in your program, it is good practice to have a corresponding **#pragma convert(pop)** as well. This will prevent one file from potentially changing the codepage of another file that is included.

#pragma convert takes precedence over **#pragma convlit**.

The following are not converted:

- A string or character constant specified in hexadecimal or octal escape sequence format (because it represents the *value* of the desired character on output).
- A string literal that is part of a `#include` or pragma directive.
- C++ String literals that are used to specify linkage (for example, `extern "C"`). C++

Related information

- "#pragma convlit"
- The LOCALE, ASCII, and CONVLIT options in the *z/OS XL C/C++ User's Guide.*

#pragma convlit

Category

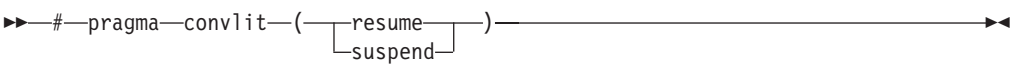
"Portability and migration" on page 536

Purpose

Suspends the string literal conversion that the CONVLIT compiler option performs during specific portions of your program.

You can then resume the conversion at some later point in the file.

Syntax



Defaults

See the *z/OS XL C/C++ User's Guide* for information about default code pages.

Parameters

suspend
Instructs the compiler to suspend all string literal conversions for the code following the directive.

resume
Instructs the compiler to resume all string literal conversions for the code following the directive.

Usage

If you use the PPONLY option, the compiler echoes the **convlit** pragma to the expanded source file.

Related information

- "#pragma convert" on page 543
- The CONVLIT option in the *z/OS XL C/C++ User's Guide.*

#pragma csect

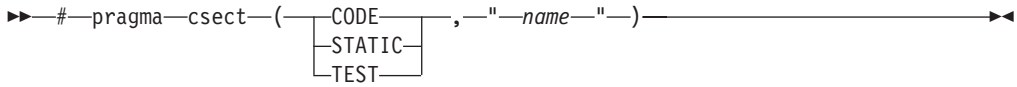
Category

"Object code control" on page 535

Purpose

Identifies the name for the code, static, or test control section (CSECT) of the object module.

Syntax



Defaults

See the CSECT option in the *z/OS XL C/C++ User's Guide.*

Parameters

CODE
Specifies the CSECT that contains the executable code (C functions) and constant data.

STATIC
Designates the CSECT that contains all program variables with the `static` storage class and all character strings.

TEST
Designates the CSECT that contains debug information. You must also specify the TEST compiler option.

name
The name that is used for the applicable CSECT. The compiler does not map the name in any way. If the name is greater than 8 characters, the LONGNAME option must be in effect and you must use the binder. The name

must not conflict with the name of an exposed name (external function or object) in a source file. In addition, it must not conflict with another **#pragma csect** directive or **#pragma map** directive. For example, the name of the code CSECT must differ from the name of the static and test CSECTs. For more information on CSECT naming rules, see the CSECT compiler option description in the *z/OS XL C/C++ User's Guide.*

Usage

At most, three **#pragma csect** directives can appear in a source program as follows:

- One for the code CSECT
- One for the static CSECT
- One for the debug CSECT

When both **#pragma csect** and the CSECT compiler option are specified, the compiler first uses the option to generate the CSECT names, and then the **#pragma csect** overrides the names generated by the option.

Examples

Suppose that you compile the following code with the option CSECT(abc) and program name `foo.c`.

```
#pragma csect (STATIC, "blah")
int main ()
{
   return 0;
}
```

First, the compiler generates the following csect names:

```
STATIC:abc#foo#S
CODE: abc#foo#C
TEST: abc#foo#T
```

Then the **#pragma csect** overrides the static CSECT name, which renders the final CSECT name to be:

```
STATIC: blah
CODE: abc#foo#C
TEST: abc#foo#T
```

IPA effects

Use the **#pragma csect** directive when naming regular objects only if the OBJECT suboption of the IPA compiler option is in effect. Otherwise, the compiler discards the CSECT names that **#pragma csect** generated.

Related information

- CSECT option in the *z/OS XL C/C++ User's Guide.*

#pragma define (C++ only)

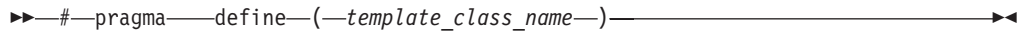
Category

Template control

Purpose

Provides an alternative method for explicitly instantiating a template class.

Syntax



Parameters

template_class_name
The name of the template class to be instantiated.

Usage

This pragma provides the equivalent functionality to C++ explicit instantiation definitions. It is provided for compatibility with earlier releases only. New applications should use C++ explicit instantiation definitions.

This pragma can appear anywhere an explicit instantiation definition can appear.

Examples

The following directive:

```
#pragma define(Array<char>)
```

is equivalent to the following explicit instantiation:

```
template class Array<char>;
```

Related information

- "Explicit instantiation" on page 456

#pragma disjoint

Category

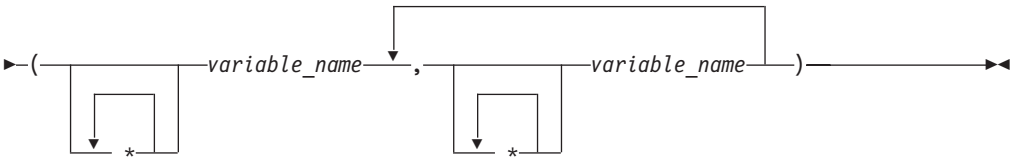
Optimization and tuning

Purpose

Lists identifiers that are not aliased to each other within the scope of their use.

By informing the compiler that none of the identifiers listed in the pragma shares the same physical storage, the pragma provides more opportunity for optimizations.

Syntax



Parameters

variable_name
The name of a variable. It must not refer to any of the following:

- A member of a structure, class, or union
- A structure, union, or enumeration tag
- An enumeration constant
- A `typedef` name
- A label

Usage

The **#pragma disjoint** directive asserts that none of the identifiers listed in the pragma share physical storage; if any the identifiers *do* actually share physical storage, the pragma may give incorrect results.

The pragma can appear anywhere in the source program that a declaration is allowed. An identifier in the directive must be visible at the point in the program where the pragma appears.

You must declare the identifiers before using them in the pragma. Your program must not dereference a pointer in the identifier list nor use it as a function argument before it appears in the directive.

Examples

The following example shows the use of **#pragma disjoint**.

```
int a, b, *ptr_a, *ptr_b;

one_function()
{
  #pragma disjoint(*ptr_a, b)   /* *ptr_a never points to b */
  #pragma disjoint(*ptr_b, a)   /* *ptr_b never points to a */

  b = 6;
  *ptr_a = 7;   /* Assignment will not change the value of b  */

  another_function(b);    /* Argument "b" has the value 6  */
}
```

External pointer `ptr_a` does not share storage with and never points to the external variable `b`. Consequently, assigning 7 to the object to which `ptr_a` points will not change the value of `b`. Likewise, external pointer `ptr_b` does not share storage with and never points to the external variable `a`. The compiler can assume that the argument to `another_function` has the value 6 and will not reload the variable from memory.

#pragma do_not_instantiate (C++ only)

Category

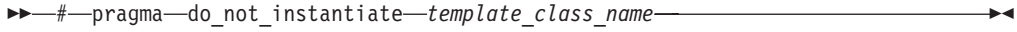
Template control

Purpose

Prevents the specified template declaration from being instantiated.

You can use this pragma to suppress the implicit instantiation of a template for which a definition is supplied.

Syntax



Parameters

template_class_name
The name of the template class that should not be instantiated.

Usage

If you are handling template instantiations manually (that is, **NOTEMPINC** and **NOTEMPLATEREGISTRY** are specified), and the specified template instantiation already exists in another compilation unit, using **#pragma do_not_instantiate** ensures that you do not get multiple symbol definitions during the link step.

C++11 **#pragma do_not_instantiate** on a class template specialization is treated as an explicit instantiation declaration of the specialization. This pragma provides a subset of the functionality of standard C++11 explicit instantiation declarations. It is provided for compatibility purposes only and is not recommended. New applications should use standard C++11 explicit instantiation declarations instead. C++11

Examples

The following shows the usage of the pragma:

```
#pragma do_not_instantiate Stack < int >
```

Related information

- "#pragma define (C++ only)" on page 546

#pragma enum

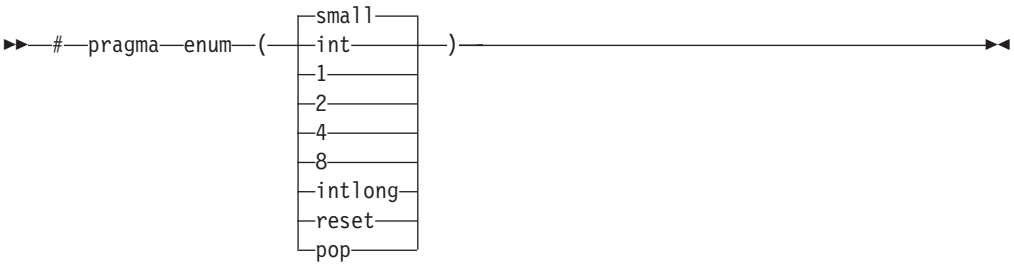
Category

"Floating point and integer control" on page 533

Purpose

Specifies the amount of storage occupied by enumerations.

Syntax



Defaults

The default is **small**.

Parameters

small
Specifies that an enumeration occupies the minimum amount of storage required by the smallest predefined type that can represent that range of enumeration constants: either 1, 2, or 4 bytes of storage. If the specified storage size is smaller than that required by the range of the enumeration constants, the compiler issues a diagnostic message. For example:

```
#pragma enum(1)
enum e_tag {
   a=0,
   b=SHRT_MAX  /* diagnostic message */
} e_var;
#pragma enum(reset)
```

int
Specifies that an enumeration occupies 4 bytes of storage and is represented by `int`.

1 Specifies that an enumeration occupies 1 byte of storage.

2 Specifies that an enumeration occupies 2 bytes of storage.

4 Specifies that an enumeration occupies 4 bytes of storage.

8 Specifies that an enumeration occupies 8 bytes of storage. This suboption is only valid with LP64.

C++ **intlong**
Specifies that an enumeration occupies 8 bytes of storage and is represented as a `long` if the range of the enumeration constants exceeds the limit for type `int`. Otherwise, enumerations occupy 4 bytes of storage and are of type `int`. This suboption is only valid with LP64. C++

reset
pop
Sets the `enum` setting to that which was in effect before the current setting.

Usage

The directive is in effect until the next valid **#pragma enum** directive is encountered. For every **#pragma enum** directive in your program, it is good

practice to have a corresponding **#pragma enum(reset)** or **#pragma enum(pop)** as well. This is the only way to prevent one file from potentially changing the enum setting of another file that is included.

You cannot have **#pragma enum** directives within the declaration of an enumeration. The following code segment generates a warning message and the second occurrence of the pragma is ignored:

```
#pragma enum(small)
enum e_tag {
     a,
     b,
#pragma enum(int) /*cannot be within a declaration */
     c
} e_var;
```

Related information

For detailed information on the preferred sign and type for each range of enumeration constants, see the description of the ENUMSIZE compiler option in the *z/OS XL C/C++ User's Guide.*

#pragma environment (C only)

Category

"Object code control" on page 535

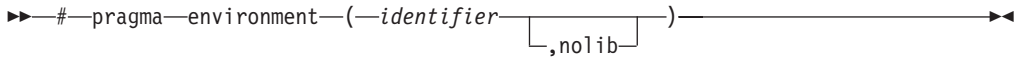
Purpose

Uses C code as an assembler substitute.

The directive allows you to do the following:

- Specify a function as an entry point other than `main`
- Omit setting up a C environment on entry to the named function
- Specify several system exits that are written in C code in the same executable

Syntax



Defaults

Not applicable.

Parameters

identifier
The name of the function that is to be the alternate entry point.

nolib
The Language Environment is established, but the LE runtime library is not loaded at run time. If you omit this argument, the library is loaded.

Usage

If you specify any other value than **nolib** after the function name, behavior is not defined.

#pragma execution_frequency

Category

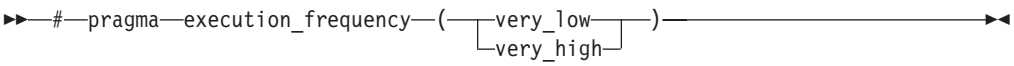
"Optimization and tuning" on page 534

Purpose

Marks program source code that you expect will be either very frequently or very infrequently executed.

When optimization is enabled, the pragma is used as a hint to the optimizer.

Syntax



Parameters

very_low
Marks source code that you expect will be executed very infrequently.

very_high
Marks source code that you expect will be executed very frequently.

Usage

Use this pragma in conjunction with an optimization option; if optimization is not enabled, the pragma has no effect.

The pragma must be placed within block scope, and acts on the closest point of branching.

Examples

In the following example, the pragma is used in an `if` statement block to mark code that is executed infrequently.

```
int *array = (int *) malloc(10000);

if (array == NULL) {
    /* Block A */
    #pragma execution_frequency(very_low)
    error();
}
```

In the next example, the code block `Block B` is marked as infrequently executed and `Block C` is likely to be chosen during branching.

```
if (Foo > 0) {
    #pragma execution_frequency(very_low)
    /* Block B */
    doSomething();
```

```
} else {
    /* Block C */
    doAnotherThing();
}
```

In this example, the pragma is used in a `switch` statement block to mark code that is executed frequently.

```
while (counter > 0) {
    #pragma execution_frequency(very_high)
    doSomething();
} /* This loop is very likely to be executed.    */

switch (a) {
    case 1:
        doOneThing();
        break;
    case 2:
        #pragma execution_frequency(very_high)
        doTwoThings();
        break;
    default:
        doNothing();
}    /* The second case is frequently chosen.   */
```

The following example shows how the pragma must be applied at block scope and affects the closest branching.

```
int a;
#pragma execution_frequency(very_low)
int b;

int foo(boolean boo) {
    #pragma execution_frequency(very_low)
    char c;

    if (boo) {
        /* Block A */
        doSomething();
        {
            /* Block C */
            doSomethingAgain();
            #pragma execution_frequency(very_low)
            doAnotherThing();
        }
    } else {
        /* Block B */
        doNothing();
    }

    return 0;
}

#pragma execution_frequency(very_low)
```

#pragma export

Category

"Object code control" on page 535

Purpose

Declares that an external function or variable is to be exported.

The pragma also specifies the name of the function or variable to be referenced outside the module. You can use this pragma to export functions or variables from a DLL module.

Syntax

#pragma export

Defaults

Not applicable.

Parameters

identifier
The name of a variable or function to be exported.

Usage

You can specify this pragma anywhere in the DLL source code, on its own line, or with other pragmas. You can also specify it before or after the definition of the variable or function. You must externally define the exported function or variable.

If the specification for a `const` variable in a **#pragma export** directive conflicts with the ROCONST option, the pragma directive takes precedence over the compiler option, and the compiler issues an informational message. The `const` variable gets exported and it is considered reentrant.

The `main` function can not be exported.

IPA effects

If you specify this pragma in your source code in the IPA compile step, you cannot override the effects of this pragma on the IPA link step.

Related reference:

"The _Export qualifier (C++ only)" on page 125
"The _Export function specifier (C++ only)" on page 233

#pragma extension

Category

"Language element control" on page 532

Purpose

Enables extended language features.

Syntax

Defaults

See the description of the LANGLVL compiler option in the *z/OS XL C/C++ User's Guide.*

Parameters

pop
Reverts the language level setting to the previous one defined for the file (if any).

Usage

The directive must only occur outside external declarations.

Multiple **#pragma extension** and **#pragma extension(pop)** directive pairs can appear in the same file. You should place a **#pragma extension(pop)** directive at the end of the section of code to which the **#pragma extension** applies within the same file; if you do not do so, the compiler will insert a pop directive.

Do not pop a **#pragma extension** directive from within a nested include file.

C When **#pragma langlvl** is embedded in **#pragma extension** and **#pragma extension (pop)** directives, an informational message is issued and **#pragma langlvl** is ignored.

Examples

The following example shows how **#pragma extension** is applied to an included file, assuming that the default language level setting is ANSI:

```
#pragma extension
#include_next <stddef.h>   /* C++: langlvl = extended; C: langlvl = extc89 */
#pragma extension(pop)
#include <pthread.h>       /* langlvl = ansi */
```

The following example shows how **#pragma extension** is applied to a section of code, assuming that the default language level setting is ANSI:

```
int alignofChar()
{
return __alignof__(char);   /* langlvl = ansi, __alignof__ is treated as an
                               identifier and a diagnostic message
                               is issued */
}
#pragma extension

int alignofInt()          /* C++: langlvl = extended; C: langlvl = extc89 */
{
return __alignof__(int);   /* __alignof__ is treated as a keyword
                              and no error message is issued */
}
```

Related information

- "#pragma langlvl (C only)" on page 564
- The LANGLVL option in the *z/OS XL C/C++ User's Guide.*

#pragma filetag

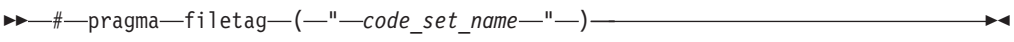
Category

"Language element control" on page 532

Purpose

Specifies the code set in which the source code was entered.

Syntax



Defaults

See the *z/OS XL C/C++ User's Guide* for information about default code pages.

Parameters

code_set_name
The name of the source code set.

Usage

Since the **#** character is variant between code sets, use the trigraph representation **??=** instead of **#** as illustrated below.

The **#pragma filetag** directive can appear only once per source file. It must appear before the first statement or directive, except for conditional compilation directives or the `#line` directive. For example:

```
??=line 42
??=ifdef COMPILER_VER         /* This is allowed. */
??=pragma filetag ("code set name")
??=endif
```

The **#pragma filetag** directive should not appear in combination with any other **#pragma** directives. For example, the following is incorrect:

```
??=pragma filetag ("IBM-1047") export (baffle_1)
```

If there are comments before the pragma, the compiler does not translate them to the code page that is associated with the LOCALE option.

Related information

- The LOCALE, ASCII, and CONVLIT options in the *z/OS XL C/C++ User's Guide.*

#pragma hashome (C++ only)

Category

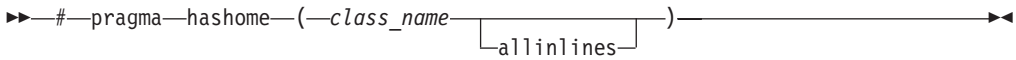
Object code control

Purpose

Informs the compiler that the specified class has a home module that will be specified by **#pragma ishome**.

This class's virtual function table, along with certain inline functions, will not be generated as static. Instead, they will be referenced as externals in the compilation unit of the class in which **#pragma ishome** is specified.

Syntax



Parameters

class_name
The name of a class to be referenced externally. *class_name* must be a class and it must be defined.

allinlines
Specifies that all inline functions from within *class_name* should be referenced as being external.

Usage

A warning will be produced if there is a **#pragma ishome** without a matching **#pragma hashome**.

Examples

In the following example, compiling the code samples will generate virtual function tables and the definition of `S::foo()` only for compilation unit `a.o`, but not for `b.o`. This reduces the amount of code generated for the application.

```
// a.h
struct S
{
   virtual void foo() {}

   virtual void bar();
};



// a.C
#pragma ishome(S)
#pragma hashome (S)

#include "a.h"

int main()
{
   S s;
   s.foo();
   s.bar();
}



// b.C
#pragma hashome(S)
#include "a.h"

void S::bar() {}
```

Related information

- "#pragma ishome (C++ only)" on page 561

#pragma implementation (C++ only)

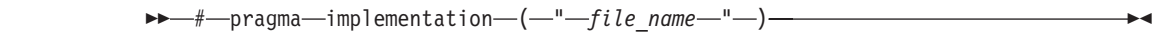
Category

Template control

Purpose

For use with the TEMPINC compiler option, supplies the name of the file containing the template definitions corresponding to the template declarations contained in a header file.

Syntax



Parameters

file_name
The name of the file containing the definitions for members of template classes declared in the header file.

Usage

This pragma is not normally required if your template implementation file has the same name as the header file containing the template declarations, and a .c extension. You only need to use the pragma if the template implementation file does not conform to this file-naming convention. For more information about using template implementation files, see "Using templates in C++ programs" in the *z/OS XL C/C++ Programming Guide.*

#pragma implementation is only effective if the TEMPINC option is in effect. Otherwise, the pragma has no meaning and is ignored.

The pragma can appear in the header file containing the template declarations, or in a source file that includes the header file. It can appear anywhere that a declaration is allowed.

Related information

- The TEMPINC option in the *z/OS XL C/C++ User's Guide.*
- "Using templates in C++ programs" in the *z/OS XL C/C++ Programming Guide.*

#pragma info (C++ only)

Category

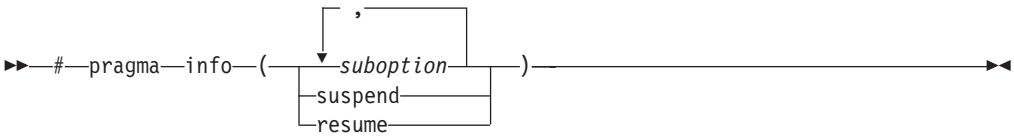
"Listings, messages and compiler information" on page 534

Purpose

Controls the diagnostic messages that are generated by the compiler.

You can use this pragma directive in place of the INFO option.

Syntax



Defaults

See the INFO option in the *z/OS XL C/C++ User's Guide.*

Parameters

suboption
Any suboption allowed by the INFO compiler option. For details, see the description of the INFO option in the *z/OS XL C/C++ User's Guide.*

suspend
Suspends the diagnostics that the pragma or INFO compiler option performs during specific portions of your program.

resume
Resumes the same level of diagnostics in effect before the **suspend** pragma was specified.

Related information

- "#pragma checkout" on page 540
- The INFO option in the *z/OS XL C/C++ User's Guide.*

#pragma inline (C only) / noinline

Category

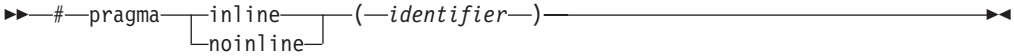
"Optimization and tuning" on page 534

Purpose

Specifies that a C function is to be inlined, or that a C or C++ function is not to be inlined.

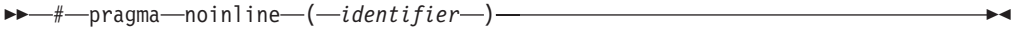
Syntax

C



C

C++



C++

Defaults

The default is **noinline**.

Parameters

inline
Inlines the named function on every call, provided you have specified the INLINE or the OPT compiler options (otherwise it has no effect). The function is inlined in both selective (NOAUTO) and automatic (AUTO) mode.

noinline

Prevents the named function from being inlined on any call, disabling any effects of the INLINE or OPT compiler options (the pragma has no effect in selective (NOAUTO) mode). It also takes precedence over the C/C++ keyword `inline`.

identifier
The name of a function to be included or excluded for inlining.

Usage

C The directive must be at file scope. C

C++ The directive can be placed anywhere. C++

IPA effects

If you use either the **#pragma inline** or the **#pragma noinline** directive in your source, you can later override them with an appropriate IPA link control file directive during the IPA link step. The compiler uses the IPA link control file directive in the following cases:

- If you specify both the **#pragma noinline** directive and the IPA link control file **inline** directive for a function.
- If you specify both the **#pragma inline** directive and the IPA link control file **noinline** directive for a function.

Related information

- "The inline function specifier" on page 228
- The INLINE option in the *z/OS XL C/C++ User's Guide.*

#pragma insert_asm (C only)

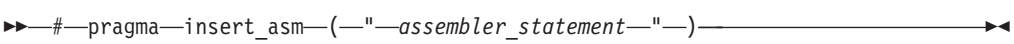
Category

"Object code control" on page 535

Purpose

Enables users to provide additional assembler statements that are inserted into the compiler-generated High-Level Assembler (HLASM) code.

Syntax



Defaults

Not applicable.

Parameters

assembler_statement
Valid assembler statement inserted into the generated HLASM code.

Usage

#pragma insert_asm can be used with a `_Pragma` operator. You must use the '\' character before double quotation marks when specifying the statements in the `_Pragma` operator; for example, `_Pragma ("insert_asm(\"MYSTATEMENT\")")`.

Multiple **insert_asm** pragmas can be specified in a program. The compiler will insert the statements into the generated HLASM in the order they are specified. The statements are inserted at the end of the compiler-generated code for the compilation unit and are placed before the END statement.

The compiler does not have any knowledge of the contents of the specified statements. You must ensure that the statements do not cause assembly errors when compiling the final HLASM code.

You must specify the METAL compiler option in order to use **#pragma insert_asm**. For more information on the METAL compiler option, see *z/OS XL C/C++ User's Guide*.

IPA effects

Not applicable.

#pragma ishome (C++ only)

Category

Object code control

Purpose

Informs the compiler that the specified class's home module is the current compilation unit.

The home module is where items, such as the virtual function table, are stored. If an item is referenced from outside of the compilation unit, it will not be generated outside its home. This can reduce the amount of code generated for the application.

Syntax

Parameters

class_name
The name of the class whose home will be the current compilation unit.

Usage

A warning will be produced if there is a **#pragma ishome** without a matching **#pragma hashome**.

Examples

See "#pragma hashome (C++ only)" on page 556

Related information

• "#pragma hashome (C++ only)" on page 556

#pragma isolated_call

Category

Optimization and tuning

Purpose

Specifies functions in the source file that have no side effects other than those implied by their parameters.

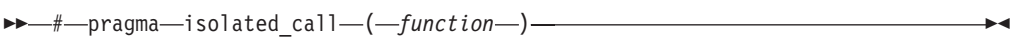
Essentially, any change in the state of the runtime environment is considered a side effect, including:

- Accessing a volatile object
- Modifying an external object
- Modifying a static object
- Modifying a file
- Accessing a file that is modified by another process or thread
- Allocating a dynamic object, unless it is released before returning
- Releasing a dynamic object, unless it was allocated during the same invocation
- Changing system state, such as rounding mode or exception handling
- Calling a function that does any of the above

Marking a function as isolated indicates to the optimizer that external and static variables cannot be changed by the called function and that pessimistic references to storage can be deleted from the calling function where appropriate. Instructions can be reordered with more freedom, resulting in fewer pipeline delays and faster execution in the processor. Multiple calls to the same function with identical parameters can be combined, calls can be deleted if their results are not needed, and the order of calls can be changed.

Syntax

Pragma syntax



Defaults

Not applicable.

Parameters

function
The name of a function that does not have side effects or does not rely on functions or processes that have side effects. *function* is a primary expression that can be an identifier, operator function, conversion function, or qualified name. An identifier must be of type function or a `typedef` of function. C++ If the name refers to an overloaded function, all variants of that function are marked as isolated calls. C++

Usage

The only side effect that is allowed for a function named in the pragma is modifying the storage pointed to by any pointer arguments passed to the function, that is, calls by reference. The function is also permitted to examine nonvolatile external objects and return a result that depends on the nonvolatile state of the runtime environment. Do not specify a function that causes any other side effects; that calls itself; or that relies on local static storage. If a function is incorrectly identified as having no side effects, the program behavior might be unexpected or produce incorrect results.

The **#pragma isolated_call** directive can be placed at any point in the source file, before or after calls to the function named in the pragma.

Predefined macros

None.

Examples

The following example shows you when to use the **#pragma isolated_call** directive (on the `addmult` function). It also shows you when not to use it (on the `same` and `check` functions):

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

int addmult(int op1, int op2);
#pragma isolated_call(addmult)

/* This function is a good candidate to be flagged as isolated as its */
/* result is constant with constant input and it has no side effects. */
int addmult(int op1, int op2) {
  int rslt;

  rslt = op1*op2 + op2;
  return rslt;
}

/* The function 'same' should not be flagged as isolated as its state */
/* (the static variable delta) can change when it is called. */
int same(double op1, double op2) {
  static double delta = 1.0;
  double temp;

  temp = (op1-op2)/op1;
  if (fabs(temp) < delta)
    return 1;
  else {
    delta = delta / 2;
    return 0;
  }
```

```
}

/* The function 'check' should not be flagged as isolated as it has a */
/* side effect of possibly emitting output. */
int check(int op1, int op2) {
  if (op1 < op2)
    return -1;
  if (op1 > op2)
    return 1;
  printf("Operands are the same.\n");
  return 0;
}
```

IPA effects

If you specify this pragma in your source code in the IPA compile step, you cannot override the effects of the pragma on the IPA link step.

#pragma langlvl (C only)

Category

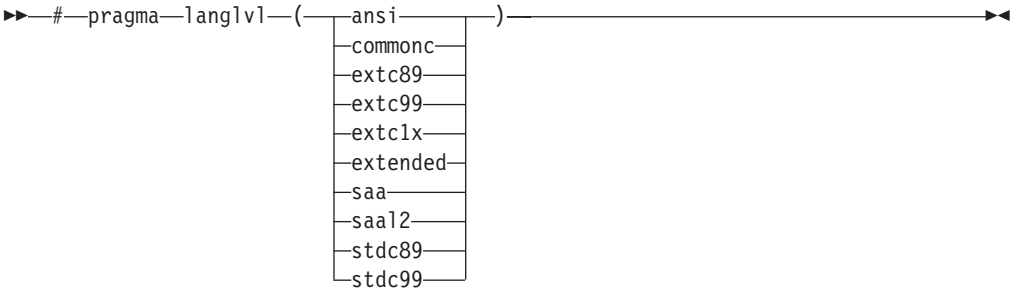
"Language element control" on page 532

Purpose

Determines whether source code and compiler options should be checked for conformance to a specific language standard, or subset or superset of a standard.

This pragma is equivalent to the LANGLVL compiler option.

Syntax



Defaults

See the LANGLVL option in the *z/OS XL C/C++ User's Guide.*

Parameters

ansi
Allows only language constructs that support the ISO C standards. It does not permit packed decimal types and issues an error message when it detects assignment between integral types and pointer types.

extc89
Allows language constructs that support the ISO C89 standard, and accepts implementation-specific language extensions.

extc99
Allows language constructs that support the ISO C99 standard, and accepts implementation-specific language extensions.

extc1x
Allows compilation that is based on the C11 standard, invoking all the C and currently-supported C11 features that are implemented.

extended
The option permits packed decimal types and it issues a warning message when it detects assignment between integral types and pointer types.

commonc
Allows compilation of code that contains constructs defined by the X/Open Portability Guide (XPG) Issue 3 C language (referred to as Common Usage C). It is roughly equivalent to K&R C.

saa
Compilation conforms to the IBM SAA C Level 2 CPI language definition.

saal2
Compilation conforms to the SAA C Level 2 CPI language definition, with some exceptions.

stdc89
Allows language constructs that support the ISO C89 standard.

stdc99
Allows language constructs that support the ISO C99 standard.

Usage

You can only specify this pragma only once in a source file, and must appear before any statements. The compiler uses predefined macros in the header files to make declarations and definitions available that define the specified language level. When both the pragma and the compiler option are specified, the compiler option takes precedence over the pragma. Note that if you would like to specify the **extc89** language level, you can also do by using **#pragma extension**.

Note: In z/OS UNIX System Services, if the **c89** environment variable {_NOCMDOPTS} is set to 1, **#pragma langlvl** has no effect. You must use the compiler option LANGLVL instead of the pragma.

Related information

- "Standard pragmas" on page 526
- "#pragma extension" on page 554
- The LANGLVL option in the *z/OS XL C/C++ User's Guide.*

#pragma leaves

Category

Optimization and tuning

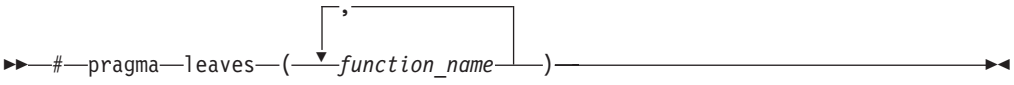
Purpose

Informs the compiler that a named function never returns to the instruction following a call to that function.

By informing the compiler that it can ignore any code after the function, the directive allows for additional opportunities for optimization.

This pragma is commonly used for custom error-handling functions, in which programs can be terminated if a certain error is encountered.

Syntax



Parameters

function_name
The name of the function that does not return to the instruction following the call to it.

Defaults

Not applicable.

Usage

If you specify the LIBANSI compiler option (which informs the compiler that function names that match functions in the C standard library are in fact C library functions), the compiler checks whether the `longjmp` family of functions (`longjmp`, `_longjmp`, `siglongjmp`, and `_siglongjmp`) contain **#pragma leaves**. If the functions do not contain this pragma directive, the compiler will insert this directive for the functions. This is not shown in the listing.

Examples

```
#pragma leaves(handle_error_and_quit)
void test_value(int value)
{
 if (value == ERROR_VALUE)
 {
  handle_error_and_quit(value);
  TryAgain(); // optimizer ignores this because
    // never returns to execute it
 }
}
```

IPA effects

If you specify the **#pragma leaves** directive in your source code in the IPA compile step, you cannot override the effects of this directive in the IPA link step.

Related information

- "#pragma reachable" on page 590.

#pragma linkage (C only)

Category

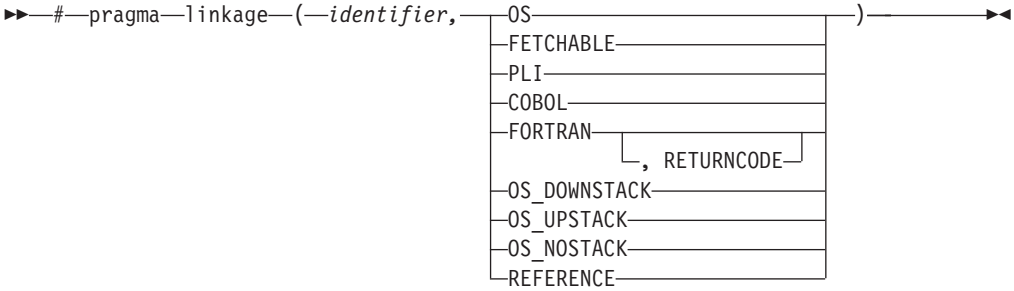
"Object code control" on page 535

Purpose

Identifies the entry point of modules that are used in interlanguage calls from C programs as well as the linkage or calling convention that is used on a function call.

The directive also designates other entry points within a program that you can use in a `fetch` operation.

Syntax



Defaults

C linkage.

Parameters

identifier
The name of a function that is to be the entry point of the module, or a typedef name that will be used to define the entry point. (See below for an example.)

FETCHABLE
Indicates that *identifier* can be used in a `fetch` operation. A fetched XPLINK module must have its entry point defined with a **#pragma linkage(..., fetchable)** directive.

OS Designates *identifier* as an OS linkage entry point. OS linkage is the basic linkage convention that is used by the operating system. If the caller is compiled with NOXPLINK, on entry to the called routine, its register 13 points to a standard Language Environment stack frame, beginning with a 72-byte save area. The stack frame is compatible with Language Environment languages that expect by-reference calling conventions and with the Language Environment-supplied assembler prologue macro. If the caller is compiled with XPLINK, the behavior depends on the OSCALL suboption of the XPLINK compiler option. This suboption selects the behavior for linkage OS from among OS_DOWNSTACK, OS_UPSTACK, and OS_NOSTACK (the default). This means that applications which use linkage OS to communicate among C or C++ functions will need source changes when recompiled with XPLINK. See the description that follows for REFERENCE.

PLI
Designates *identifier* as a PL/I linkage entry point.

COBOL
Designates *identifier* as a COBOL linkage entry point.

FORTRAN
Designates *identifier* as a FORTRAN linkage entry point.

RETURNCODE indicates to the compiler that the routine named by *identifier* is a FORTRAN routine, which returns an alternate return code. It also indicates that the routine is defined outside the translation unit. You can retrieve the return code by using the `fortrc` function. If the compiler finds the function definition inside the translation unit, it issues an error message. Note that you can define functions outside the translation unit, even if you do not specify the RETURNCODE keyword.

OS_DOWNSTACK
Designates *identifier* as an OS linkage entry point in XPLINK mode with a downward growing stack frame.

If the function identified by *identifier* is defined within the translation unit and is compiled using the NOXPLINK option, the compiler issues an error message.

OS_UPSTACK
Designates *identifier* as an OS linkage entry point in XPLINK mode with a traditional upward growing stack frame.

This linkage is required for a new XPLINK downward-stack routine to be able to call a traditional upward-stack OS routine. This linkage explicitly invokes compatibility code to swap the stack between the calling and the called routines.

If the function identified by *identifier* is defined within the translation unit and is compiled using the XPLINK option, the compiler issues an error message. Typically, the *identifier* will not be defined in a compilation. This is acceptable. In this case, it is a reference to an external procedure that is separately compiled with NOXPLINK.

OS_NOSTACK
Designates *identifier* as an OS linkage entry point in XPLINK mode with no preallocated stack frame. An argument list is constructed containing the addresses of the actual arguments. The size of the address is 4-byte for 31-bit and 8-byte for 64-bit. Register 1 is set to point to this argument list. For 31-bit only, the last item in this list has its high order bit set. For integer type arguments, the value passed is widened to the size of the `int` type, that is 4-byte. Register 13 points to a save area that may not be followed by z/OS Language Environment control structures, such as the NAB. The size of the save area is 72-byte for 31-bit and 144-byte for 64-bit. Register 14 contains the return address. Register 15 contains the entry point of the called function.

REFERENCE
This is synonymous with OS_UPSTACK in non-XPLINK mode and synonymous with OS_DOWNSTACK in XPLINK mode. Unlike the linkage OS, this is not affected by the OSCALL suboption of XPLINK. Consider using this option instead to make the source code portable between XPLINK and non-XPLINK

Usage

You can use a typedef in a **#pragma linkage** directive to associate a specific linkage convention with a function type. In the following example, the directive associates the OS linkage convention with the typedef `func_t`:

```
typedef void func_t(void);
#pragma linkage (func_t,OS)
```

This typedef can then be used in C declarations wherever a function should have OS linkage. In the following example:

```
func_t myfunction;
```

`myfunction` is declared as having type `func_t`, which is associated with OS linkage; `myfunction` would therefore have OS linkage.

Related information

- The XPLINK option in the *z/OS XL C/C++ User's Guide.*

#pragma longname/nolongname

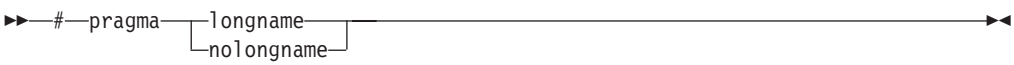
Category

"Object code control" on page 535

Purpose

Specifies whether the compiler is to generate mixed-case names that can be longer than 8 characters in the object module.

Syntax



Defaults

- C **nolongname** C
- C++ **longname** C++

Parameters

`longname`
The compiler generates mixed-case names in the object module. Names can be up to 1024 characters in length.

`nolongname`
The compiler generates truncated and uppercase names in the object module. C++ Only functions that do not have C++ linkage are given truncated and uppercase names. C++

Usage

If you use the **#pragma longname** directive, you must either use the binder to produce a program object in a PDSE, or you must use the prelinker. The binder, IPA link step, and prelinker support the long name directory that is generated by the Object Library utility for autocall.

If you specify the NOLONGNAME or LONGNAME compiler option, the compiler ignores the **#pragma longname** directive. If you specify either **#pragma nolongname** or the NOLONGNAME compiler option, and this results in mapping of two different source code names to the same object code name, the compiler will not issue an error message.

C If you have more than one preprocessor directive, **#pragma longname** may be preceded only by **#pragma filetag**, **#pragma chars**, **#pragma langlvl**, and **#pragma target**. Some directives, such as **#pragma variable** and **#pragma linkage**, are sensitive to the name handling. C

C++ You must specify **#pragma longname/nolongname** before any code. Otherwise, the compiler issues a warning message. **#pragma nolongname** must also precede any other preprocessing directive. C++

IPA effects

You must specify either the LONGNAME compile option or the **#pragma longname** preprocessor directive for the IPA compile step (unless you are using the **c89** or **cc** utility from z/OS UNIX System Services, both of which already specify the LONGNAME compiler option). Otherwise, you will receive an unrecoverable compiler error.

Related information

- "#pragma map"

#pragma map

Category

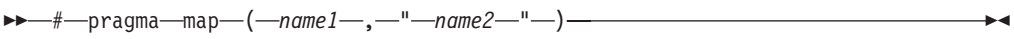
Object code control

Purpose

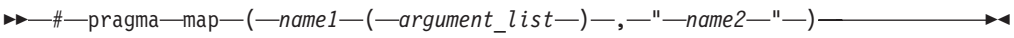
Converts all references to an identifier to another, externally defined identifier.

Syntax

#pragma map syntax (C only)



#pragma map syntax (C++ only)



Parameters

name1

The name used in the source code. C *name1* can represent a data object or function with external linkage. C C++ *name1* can represent a data object, a non-overloaded or overloaded function, or overloaded operator, with external linkage. C++ If the name to be mapped is not in the global namespace, it must be fully qualified.

name1 should be declared in the same compilation unit in which it is referenced, but should not be defined in any other compilation unit. *name1* must not be used in another **#pragma map** directive anywhere in the program.

C++ *argument_list*

The list of arguments for the overloaded function or operator function designated by *name1*. If *name1* designates an overloaded function, the function

must be parenthesized and must include its argument list if it exists. If *name1* designates a non-overloaded function, only *name1* is required, and the parentheses and argument list are optional. C++

name2

The name that will appear in the object code. C *name2* can represent a data object or function with external linkage. The compiler preserves mixed-case names. If the name is longer than 8 characters, you must use the binder and specify the LONGNAME compiler option. C

C++ *name2* can represent a data object, a non-overloaded or overloaded function, or overloaded operator, with external linkage. If the name is longer than 8 characters you must use the binder. *name2* must be specified using its mangled name. To obtain C++ mangled names, compile your source to object files only, using the **-c** compiler option, and use the **nm** operating system command on the resulting object file. C++

If the name exceeds 65535 bytes, an informational message is emitted and the pragma is ignored.

name2 may or may not be declared in the same compilation unit in which *name1* is referenced, but must not be defined in the same compilation unit. Also, *name2* should not be referenced anywhere in the compilation unit where *name1* is referenced. *name2* must not be the same as that used in another **#pragma map** directive or **#pragma csect** directive in the same compilation unit. The map *name* is not affected by the CONVLIT or the ASCII compiler options.

Usage

The **#pragma map** directive can appear anywhere in the program. Note that in order for a function to be actually mapped, the map target function (*name2*) must have a definition available at link time (from another compilation unit), and the map source function (*name1*) must be called in your program.

C++ **#pragma map** will look at the declarations prior to it and if satisfied will map the declaration. If it is not satisfied, it will check all declarations at the end of the compilation to see if it can find a match. C++

You cannot use **#pragma map** with compiler built-in functions.

Examples

The following is an example of **#pragma map** used to map a function name (using the mangled name for the map name in C++):

```
/* Compilation unit 1: */

#include <stdio.h>

void foo();
extern void bar(); /* optional */

#if __cplusplus
#pragma map (foo, "bar__Fv")
#else
#pragma map (foo, "bar")
#endif
int main()
{
foo();
```

```
}

/* Compilation unit 2: */

#include <stdio.h>

void bar()
{
printf("Hello from foo bar!\n");
}
```

The call to `foo` in compilation unit 1 resolves to a call to `bar`:

```
Hello from foo bar!
```

#pragma margins/nomargins

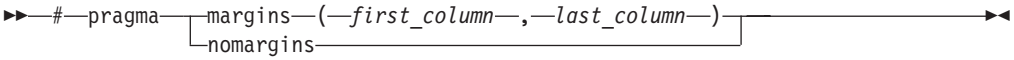
Category

"Language element control" on page 532

Purpose

Specifies the columns in the input line to scan for input to the compiler.

Syntax



Defaults

- C **margins(1,72)** for fixed-length records. **nomargins** for variable-length records. C
- C++ **nomargins** for all records. C++

Parameters

`margins`
Specifies that only text within a range of margins, specified by *first_column* and *last_column*, is to be scanned by the compiler. The compiler ignores any text in the source input that does not fall within the range.

C++ Specifying **margins** with no parameters is equivalent to **margins(1,72)** for both fixed or variable length records. C++

`nomargins`
Specifies that *all* input is to be scanned by the compiler.

`first_column`
A number representing the first column of the source input to be scanned. The value of *first_column* must be greater than 0, less than 32761, and less than or equal to the value of *last_column*.

`last_column`
A number representing the last column of the source input to be scanned. The value of *last_column* must be greater than the value of *first_column*, and less than 32761.

You can also use an asterisk (*) to indicate the last column of the input record. For example, if you specify **#pragma margins (8, *)**, the compiler scans from column 8 to the end of input record.

Usage

The pragmas override the MARGINS or NOMARGINS compiler options. The setting specified by the **#pragma margins** directive applies only to the source file or include file in which it is found. It has no effect on other include files.

You can use **#pragma margins** and **#pragma sequence** together. If they reserve the same columns, **#pragma sequence** has priority and it reserves the columns for sequence numbers. For example, assume columns 1 to 20 are reserved for the margin, and columns 15 to 25 are reserved for sequence numbers. In this case, the margin will be from column 1 to 14, and the columns reserved for sequence numbers will be from 15 to 25.

Related information

- "#pragma sequence" on page 594
- The MARGINS compiler option in the *z/OS XL C/C++ User's Guide.*

#pragma namemangling (C++ only)

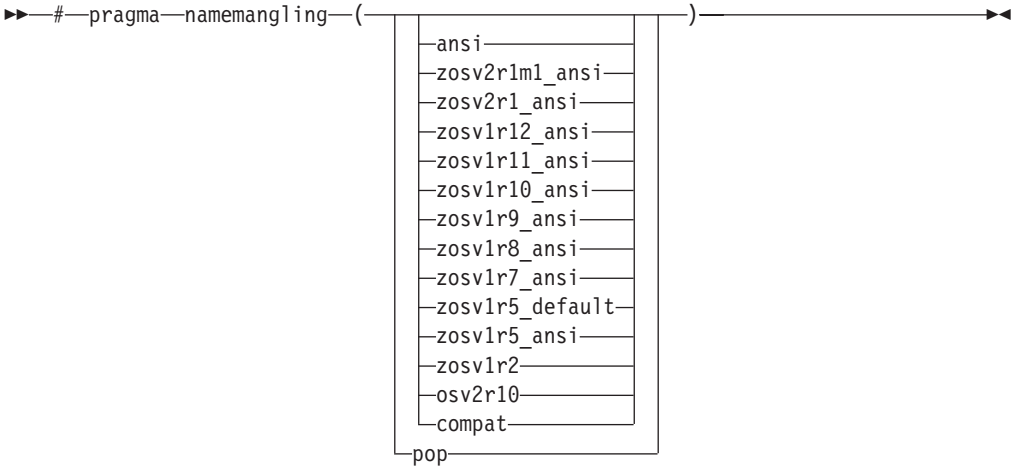
Category

"Portability and migration" on page 536

Purpose

Chooses the name mangling scheme for external symbol names generated from C++ source code.

Syntax



Defaults

See the **NAMEMANGLING** option in the *z/OS XL C/C++ User's Guide*.

Parameters

ansi
Indicates that the name mangling scheme complies with the most recent C++ language features. This setting is equivalent to **zosv2r1m1_ansi**.

zosv2r1m1_ansi
Use this scheme for compatibility with link modules from z/OS C++ V2R1M1 that were created with the **#pragma namemangling(ansi)** directive or with the NAMEMANGLING(ANSI) compiler option in effect.

zosv2r1_ansi
Use this scheme for compatibility with link modules from z/OS C++ Version 2 Release 1 that were created with the **#pragma namemangling(ansi)** directive or with the NAMEMANGLING(ANSI) compiler option in effect.

zosv1r12_ansi
Use this scheme for compatibility with link modules from z/OS C++ Version 1 Release 12 that were created with the **#pragma namemangling(ansi)** directive or with the NAMEMANGLING(ANSI) compiler option in effect.

zosv1r11_ansi
Use this scheme for compatibility with link modules from z/OS C++ Version 1 Release 11 that were created with the **#pragma namemangling(ansi)** directive or with the NAMEMANGLING(ANSI) compiler option in effect.

zosv1r10_ansi
Use this scheme for compatibility with link modules from z/OS C++ Version 1 Release 10 that were created with the **#pragma namemangling(ansi)** directive or with the NAMEMANGLING(ANSI) compiler option in effect.

zosv1r9_ansi
Use this scheme for compatibility with link modules from z/OS C++ Version 1 Release 9 that were created with the **#pragma namemangling(ansi)** directive or with the NAMEMANGLING(ANSI) compiler option in effect.

zosv1r8_ansi
Use this scheme for compatibility with link modules from z/OS C++ Version 1 Release 8 that were created with the **#pragma namemangling(ansi)** directive or with the NAMEMANGLING(ANSI) compiler option in effect.

zosv1r7_ansi
Use this scheme for compatibility with link modules from z/OS C++ Version 1 Release 7 that were created with the **#pragma namemangling(ansi)** directive or with the NAMEMANGLING(ANSI) compiler option in effect.

zosv1r5_ansi
Use this scheme for compatibility with link modules from z/OS C++ Version 1 Release 5 that were created with the **#pragma namemangling(ansi)** directive or with the NAMEMANGLING(ANSI) compiler option in effect.

zosv1r5_default
Use this scheme for compatibility with link modules from z/OS C++ Version 1 Release 5 that were created with the default mangling for that compiler. This suboption uses the same name mangling scheme as modules created with z/OS C++ Version 1 Release 2 that were created with the **#pragma namemangling(ansi)** directive or with the NAMEMANGLING(ANSI) compiler option in effect.

zosv1r2
Same semantics as the **zosv1r5_default** suboption: instructs the compiler that

the name mangling scheme is compatible with z/OS V1R2 link modules that were created with NAMEMANGLING(ANSI).

osv2r10
Use this scheme for compatibility with link modules created with the IBM OS/390® C++ Version 2 Release 10 compiler or earlier versions, or with link modules that were created with the **#pragma namemangling(compat)** directive or with the NAMEMANGLING(COMPAT) compiler option in effect.

compat
Same semantics as the **osv2r10** suboption: instructs the compiler that the name mangling scheme is compatible with that in OS/390 V2R10 and earlier versions.

pop
Restores the name mangling scheme to that which was in effect immediately before the current setting. If no previous name mangling scheme was specified in the file, the scheme specified by the NAMEMANGLING compiler option is used.

Usage

For every **#pragma namemangling** directive in your program, it is good practice to have a corresponding **#pragma namemangling(pop)** as well. In this way, it is possible to prevent one file from potentially changing the name mangling setting of another file that is included.

Related information

- "#pragma namemanglingrule (C++ only)"
- The NAMEMANGLING option in the *z/OS XL C/C++ User's Guide.*

#pragma namemanglingrule (C++ only)

Category

Portability and migration

Purpose

Provides fined-grained control over the name mangling scheme in effect for selected portions of source code, specifically with respect to the mangling of cv-qualifiers in function parameters.

When a function name is mangled, repeated function arguments of the same type are encoded according to the following compression scheme:

```
parameter → T param number [_]   #single repeat of a previous parameter
          → N repetition digit param number [_]    #2 to 9 repetitions
```

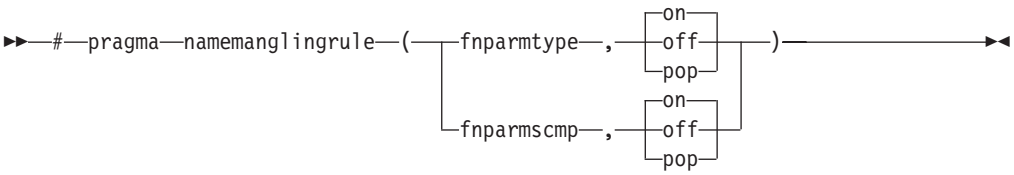
where:

param number
Indicates the number of the previous parameter which is repeated. It is followed by an underscore (_) if *param number* contains multiple digits.

repetition digit
Must be greater than 1 and less than 10. If an argument is repeated more than 9 times, this rule is applied multiple times. For example, a sequence of 38 parameters that are the same as parameter 1 mangles to `N91N91N91N91N21`.

The **#pragma namemanglingrule** directive allows you to control whether top-level cv-qualifiers are mangled in function parameters.

Syntax



Defaults

- **fnparmtype, on** when the **#pragma namemangling(ansi)** directive or the NAMEMANGLING(ANSI) compiler option is in effect. Otherwise the default is **fnparmtype,off**.
- **fnparmscmp, on** when the **#pragma namemangling(ansi)** directive or the NAMEMANGLING(ANSI) compiler option is in effect. Otherwise the default is **fnparmscmp,off**.

Parameters

fnparmtype, on
Top-level cv-qualifiers are not encoded in the mangled name of a function parameter. Also, top-level cv-qualifiers are ignored when repeated function parameters are compared for equivalence; function parameters that differ only by the use of a top-level cv-qualifier are considered equivalent and are mangled according to the compressed encoding scheme.

fnparmtype, off
Top-level cv-qualifiers are encoded in the mangled name of a function parameter. Also, repeated function parameters that differ by the use of cv-qualifiers are not considered equivalent and are mangled as separate parameters. This setting is compatible with z/OS C++ Version 1 Release 5.

fnparmtype, pop
Reverts to the previous **fnparmtype** setting in effect. If no previous settings are in effect, the default **fnparmtype** setting is used.

Note: This pragma fixes function signature ambiguities in 32-bit mode, but it is not needed in 64-bit mode since those ambiguities do not exist in 64-bit mode.

fnparmscmp, on
Intermediate-level cv-qualifiers are considered when repeated function parameters are compared for equivalence; repeated function parameters that differ by the use of intermediate-level cv-qualifiers are mangled as separate parameters.

fnparmscmp, off
Intermediate-level cv-qualifiers are ignored when repeated function parameters are compared for equivalence; function parameters that differ only by the use of an intermediate-level cv-qualifier are considered equivalent and are mangled according to the compressed encoding scheme. This setting is compatible with z/OS C++ Version 1 Release 5.

fnparmscmp, pop
Reverts to the previous **fnparmscmp** setting in effect. If no previous settings are in effect, the default **fnparmscmp** setting is used.

Usage

#pragma namemanglingrule is allowed in global, class, and function scopes. It has no effect on a block scope function declaration with external linkage.

Different pragma settings can be specified in front of function declarations and definitions. If **#pragma namemanglingrule** settings in subsequent declarations and definitions conflict, the compiler ignores those settings and issues a warning message.

Examples

The following tables show the effects of this pragma applied to different function signatures.

Table 43. Mangling of function parameters with top-level cv-qualifiers

	Mangled name	
Source name	**fnparmtype, off**	**fnparmtype, on**
void foo (const int)	foo__FCi	foo__Fi
void foo (int* const)	foo__FCPi	foo__FPi
void foo (int** const)	foo__FCPPi	foo__FPPi
void foo (int, const int)	foo__FiCi	foo__FiT1

Table 44. Mangling of function parameters with intermediate level cv-qualifiers

	Mangled name	
Source name	**fnparmscmp, on**	**fnparmscmp, off**
void foo (int** a, int* const * b)	foo__FPPiPCPi	foo__FPPiT1
void bar (int* const* a, int** b)	bar__FPCPiPPi	bar__FPCPiT1

Table 45. Mangling of function parameters with top-level and intermediate-level cv-qualifiers

	Mangled name			
Source name	**fnparmscmp, on fnparmtype, on**	**fnparmscmp, off fnparmtype, on**	**fnparmscmp, on fnparmtype, off**	**fnparmscmp, off fnparmtype, off**
void foo (int** const, int* const *)	foo__FPPiPCPi	foo__FPPiT1	foo__FCPPiPCPi	foo__FPPiT1

Related information

- "#pragma namemangling (C++ only)" on page 573
- The NAMEMANGLING option in the *z/OS XL C/C++ User's Guide.*

#pragma object_model (C++ only)

Category

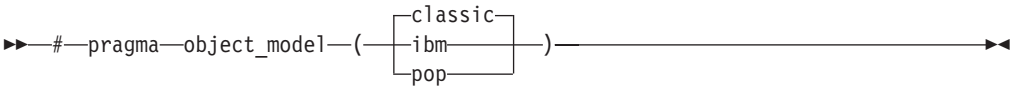
"Portability and migration" on page 536

Purpose

Sets the object model to be used for structures, unions, and classes.

The object models differ in the areas of layout for the virtual function table, support for virtual base classes, and name mangling scheme.

Syntax



Defaults

The default is **classic**.

Parameters

classic
Is compatible with name mangling and the virtual function table that was available with the previous releases of the z/OS C++ compiler.

Note: The parameter **compat** is changed to **classic**, but **compat** is still accepted as the synonym of **classic**.

ibm
Provides improved performance. Class hierarchies with many virtual base classes can benefit from this option because the size of the derived class is smaller and access to the virtual function table is faster. You must use XPLINK when specifying this option.

pop
Sets the object model setting to that which was in effect before the current setting.

Usage

Classes implicitly inherit the object model of their parent, overriding any local object model specification. All classes in the same inheritance hierarchy must have the same object model.

Related information

- The OBJECTMODEL option in the *z/OS XL C/C++ User's Guide.*

#pragma operator_new (C++ only)

Category

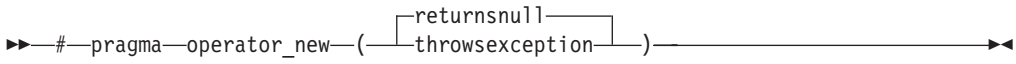
Error checking and debugging

Purpose

Determines whether the `new` and `new[]` operators throw an exception if the requested memory cannot be allocated.

This pragma is equivalent to the LANGLVL(NEWEXCP) option.

Syntax



Defaults

returnsnull

Parameters

returnsnull
If the memory requested by the `new` operator cannot be allocated, the compiler returns 0, the null pointer. Use this option for compatibility with versions of the XL C++ compiler previous to V1R7.

throwsexception
If the memory requested by the `new` operator cannot be allocated, the compiler throws a standard exception of type `std::bad_alloc`. Use this option in new applications, for conformance with the C++ standard.

Usage

The pragma can be specified only once in a source file. It must appear before any statements in the source file. This pragma takes precedence over the LANGLVL(NEWEXCP) compiler option.

Restrictions

This pragma applies only to versions of the `new` operator that throw exceptions; it does not apply to the `nothrow` or empty `throw` versions of the `new` operator (for the prototypes of all the `new` operator versions, see the description of the <new> header in the *Standard C++ Library Reference*). It also does not apply to class-specific `new` operators, user-defined `new` operators, and `new` operators with placement arguments.

Related information

- "new expressions (C++ only)" on page 185
- "Allocation and deallocation functions (C++ only)" on page 253
- The LANGLVL compiler option in the *z/OS XL C/C++ User's Guide.*

#pragma option_override

Category

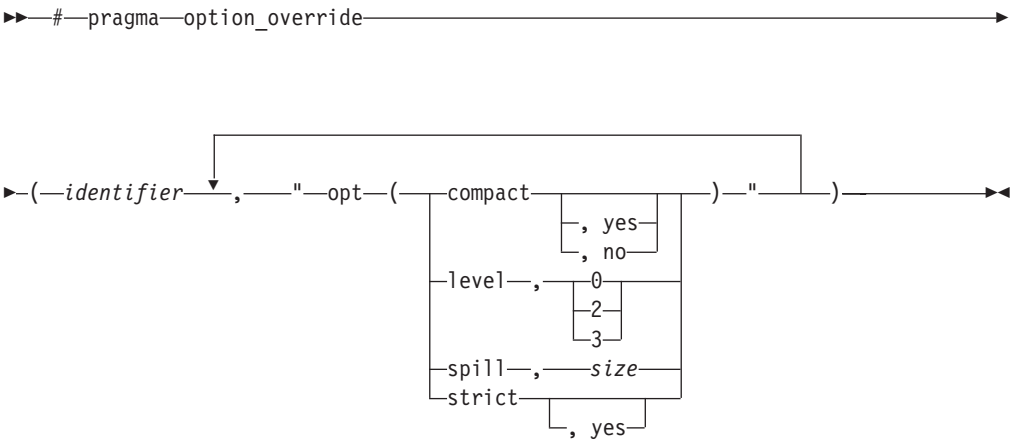
Optimization and tuning

Purpose

Allows you to specify optimization options at the subprogram level that override optimization options given on the command line.

This enables finer control of program optimization, and can help debug errors that occur only under optimization.

Syntax



Parameters

identifier
The name of a function for which optimization options are to be overridden.

The following table shows the equivalent command line option for each pragma suboption.

#pragma option_override value	Equivalent compiler option
compact	COMPACT
compact, yes	
compact, no	NOCOMPACT
level, 0	OPT(0)
level, 1	OPT(2)
level, 2	OPT(2)
level, 3	OPT(3)
spill, *size*	SPILL(*size*)
strict	STRICT
strict, yes	
strict, no	NOSTRICT

Notes:

1. If optimization level **-O3** or higher is specified on the command line, `#pragma option_override(identifier, "opt(level, 0)")` or `#pragma option_override(identifier, "opt(level, 2)")` does not turn off the implication of the **-qhot** and **-qipa** options.

Defaults

See the descriptions in the *z/OS XL C/C++ User's Guide* for the options listed in the table above for default settings.

Usage

The pragma takes effect only if optimization is already enabled by a command-line option. You can only specify an optimization level in the pragma *lower* than the level applied to the rest of the program being compiled.

The **#pragma option_override** directive only affects functions that are defined in the same compilation unit. The pragma directive can appear anywhere in the translation unit. That is, it can appear before or after the function definition, before or after the function declaration, before or after the function has been referenced, and inside or outside the function definition.

C++ This pragma cannot be used with overloaded member functions.

Examples

Suppose you compile the following code fragment containing the functions `foo` and `faa` using OPT(2). Since it contains the `#pragma option_override(faa, "opt(level, 0)")`, function `faa` will not be optimized.

```
foo(){
     .
     .
     .
     }

#pragma option_override(faa, "opt(level, 0)")

faa(){
     .
     .
     .
     }
```

IPA effects

You cannot specify the IPA compiler option for **#pragma option_override**.

During IPA compile processing, subprogram-specific options will be used to control IPA compile-time optimizations.

During IPA link processing, subprogram-specific options will be used to control IPA link-time optimizations, as well as program partitioning. They will be retained, even if the related IPA link command line option is specified.

Related information

- The OPT, COMPACT, SPILL and STRICT options in the *z/OS XL C/C++ User's Guide*.

#pragma options (C only)

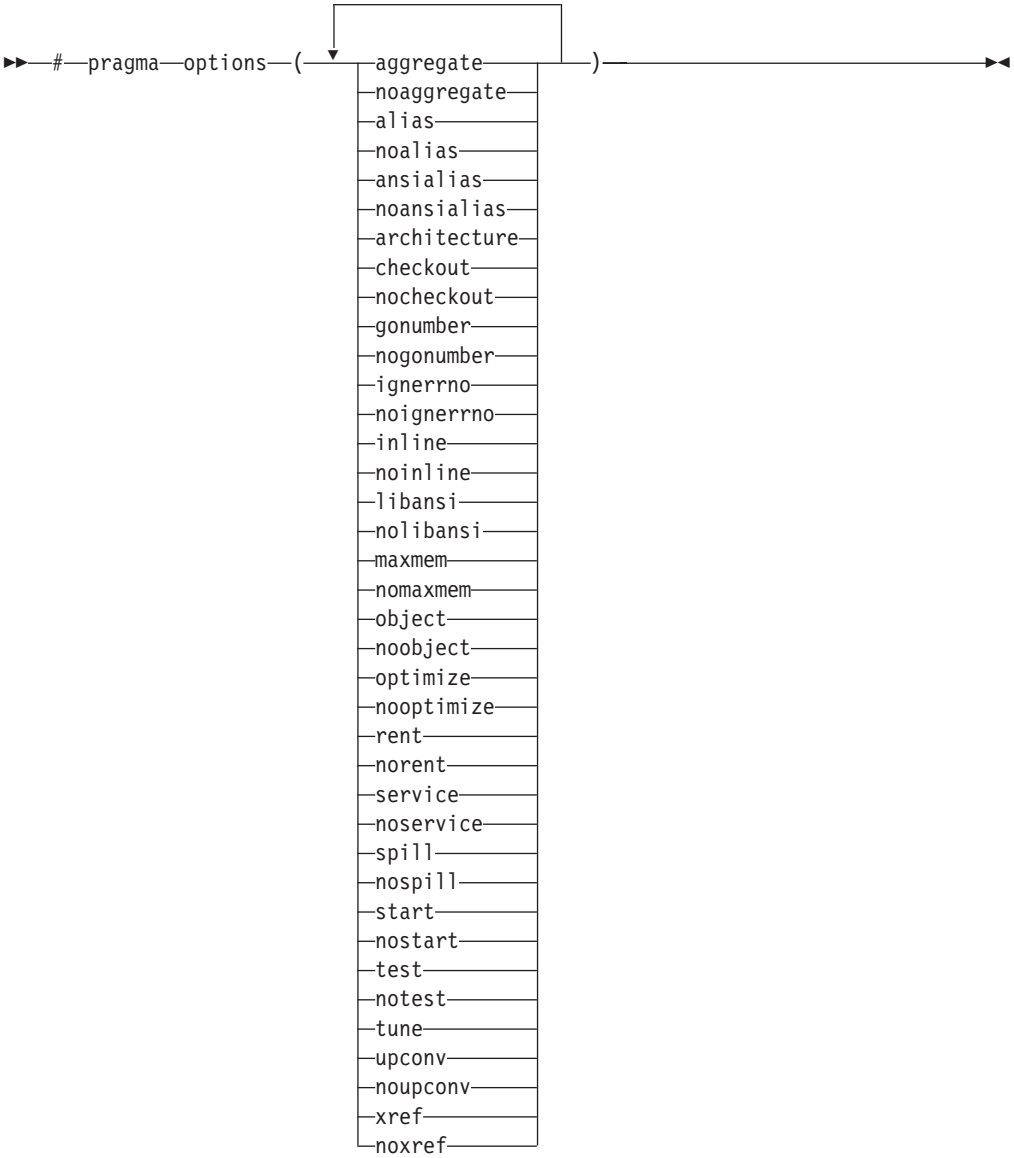
Category

"Language element control" on page 532

Purpose

Specifies a list of compiler options that are to be processed as if you had typed them on the command line or on the CPARM parameter of the IBM-supplied catalogued procedures.

Syntax



Defaults

See the *z/OS XL C/C++ User's Guide* for the default settings for these options.

Parameters

See the *z/OS XL C/C++ User's Guide* for descriptions of these options.

Usage

If you use a compile option that contradicts the options that are specified on the **#pragma options** directive, the compiler option overrides the options on the **#pragma options** directive.

If you specify an option more than once, the compiler uses the last one you specified.

IPA effects

You cannot specify the IPA compiler option for **#pragma options**.

Related information

- *z/OS XL C/C++ User's Guide*
- For a detailed description of the interaction between the INLINE compiler option on the invocation line and the **#pragma options** preprocessor directive, see the INLINE compiler option description in the *z/OS XL C/C++ User's Guide.*

#pragma pack

Category

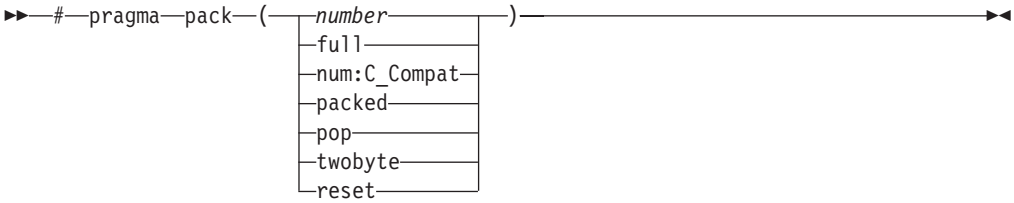
Object code control

Purpose

Sets the alignment of all aggregate members to a specified byte boundary.

If the byte boundary number is smaller than the natural alignment of a member, padding bytes are removed, thereby reducing the overall structure or union size.

Syntax



Defaults

Members of aggregates (structures, unions, and classes) are aligned on their natural boundaries and a structure ends on its natural boundary. The alignment of an aggregate is that of its strictest member (the member with the largest alignment requirement).

Parameters

number
is one of the following:

1 Aligns structure members on 1-byte boundaries, or on their natural alignment boundary, whichever is less.

2 Aligns structure members on 2-byte boundaries, or on their natural alignment boundary, whichever is less.

4 Aligns structure members on 4-byte boundaries, or on their natural alignment boundary, whichever is less.

8 Reserved for possible future use.

16 Reserved for possible future use.

full
Aligns structure members on 4-byte boundaries, or on their natural alignment boundary, whichever is less. This is the same as **#pragma pack(4)**.

num:C_Compat
num is one of the following:

- 1 (aligns structure members on 1-byte boundaries, or on their natural alignment boundary, whichever is less)
- 2 (aligns structure members on 2-byte boundaries, or on their natural alignment boundary, whichever is less)

C++

num:C_Compat aligns structure members so that the class layout will be compatible with the layout produced by the XL C compiler. This applies when:

- You have a class layout that is guarded by **#pragma pack(1)** or **#pragma pack(2)**.
- Your class data member contains a bit field that has an alignment that exceeds the alignment defined by **#pragma pack(1)** or **#pragma pack(2)**.
- The bit length of the bit field exceeds the maximum number that the bit field type is allowed to contain.

C++

packed
Aligns structure members on 1-byte boundaries, or on their natural alignment boundary, whichever is less. This is the same as **#pragma pack(1)**.

pop
Removes the previous value added with **#pragma pack**. Specifying **#pragma pack()** with no parameters is equivalent to specifying C++ **#pragma pack(pop)**

C++

or C **#pragma pack(4)** C .

Note: #pragma pack() with no parameters is deprecated. Use **#pragma pack(4)** instead.

twobyte
Aligns structure members on 2-byte boundaries, or on their natural alignment boundary, whichever is less. This is the same as **#pragma pack(2)**.

reset
Sets the packing rule to that which was in effect before the current setting.

Usage

The **#pragma pack** directive applies to the definition of an aggregate type, rather than to the declaration of an instance of that type; it therefore automatically applies to all variables declared of the specified type.

The **#pragma pack** directive modifies the current alignment rule for only the members of structures whose declarations follow the directive. It does not affect the alignment of the structure directly, but by affecting the alignment of the members of the structure, it may affect the alignment of the overall structure.

The **#pragma pack** directive cannot increase the alignment of a member, but rather can decrease the alignment. For example, for a member with data type of `short`, a **#pragma pack(1)** directive would cause that member to be packed in the structure on a 1-byte boundary, while a **#pragma pack(4)** directive would have no effect.

The **#pragma pack** directive applies only to complete declarations of structures or unions; this excludes forward declarations, in which member lists are not specified. For example, in the following code fragment, the alignment for `struct S` is 4, since this is the rule in effect when the member list is declared:

```
#pragma pack(1)
struct S;
#pragma pack(4)
struct S { int i, j, k; };
```

A nested structure has the alignment that precedes its declaration, not the alignment of the structure in which it is contained, as shown in the following example:

```
#pragma pack (4)                // 4-byte alignment
         struct nested {
           int  x;
           char y;
           int  z;
         };

         #pragma pack(1)           // 1-byte alignment
         struct packedcxx{
            char   a;
            short  b;
            struct nested  s1;     // 4-byte alignment
         };
```

If more than one **#pragma pack** directive appears in a structure defined in an inlined function, the **#pragma pack** directive in effect at the beginning of the structure takes precedence.

Examples

The following example shows how the **#pragma pack** directive can be used to set the alignment of a structure definition:

```
//  header file file.h

   #pragma pack(1)

   struct jeff{           //    this structure is packed
     short bill;          //    along 1-byte boundaries
     int *chris;
   };
   #pragma pack(reset)       //   reset to previous alignment rule
```

```
// source file anyfile.c

   #include "file.h"

   struct jeff j;          //   uses the alignment specified
                           //   by the pragma pack directive
                           //   in the header file and is
                           //   packed along 1-byte boundaries
```

This example shows how a **#pragma pack** directive can affect the size and mapping of a structure:

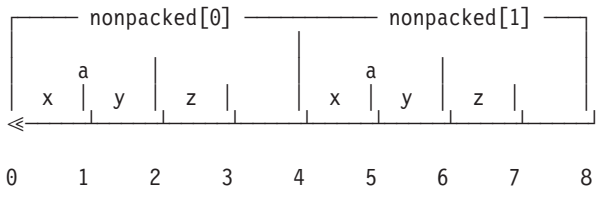
```
struct s_t {
 char a;
 int b;
 short c;
 int d;
}S;
```

Default mapping:	**With #pragma pack(1):**
size of s_t = 16	size of s_t = 11
offset of a = 0	offset of a = 0
offset of b = 4	offset of b = 1
offset of c = 8	offset of c = 5
offset of d = 12	offset of d = 7
alignment of a = 1	alignment of a = 1
alignment of b = 4	alignment of b = 1
alignment of c = 2	alignment of c = 1
alignment of d = 4	alignment of d = 1

The following example defines a union uu containing a structure as one of its members, and declares an array of 2 unions of type uu:

```
union uu {
  short    a;
  struct {
    char x;
    char y;
    char z;
  } b;
};

union uu nonpacked[2];
```

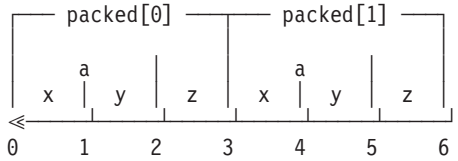
Since the largest alignment requirement among the union members is that of short a, namely, 2 bytes, one byte of padding is added at the end of each union in the array to enforce this requirement:



The next example uses **#pragma pack(1)** to set the alignment of unions of type uu to 1 byte:

```
#pragma pack(1)

 union uu {
   short    a;
   struct {
     char x;
     char y;
     char z;
   } b;
 };

 union uu pack_array[2];
```

Now, each union in the array packed has a length of only 3 bytes, as opposed to the 4 bytes of the previous case:



Related reference:

"The _Packed qualifier (C only)" on page 124

#pragma page (C only)

Category

"Listings, messages and compiler information" on page 534

Purpose

Specifies that the code following the pragma begins at the top of the page in the generated source listing.

Syntax

Defaults

Not applicable.

Parameters

number

The number of pages from the current page on which to begin writing the line of source code that follows the pragma. **#pragma page()** is the same as **#pragma page(1)**: the source line that follows the pragma will start on a new page. If you specify **#pragma page(2)**, the listing will skip one blank page and the source line following the pragma will start on the second page after the current page. In all cases, the listing continues.

#pragma pagesize (C only)

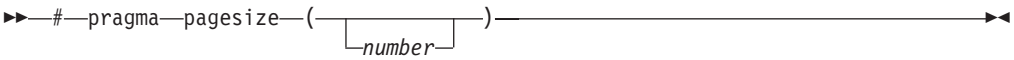
Category

"Listings, messages and compiler information" on page 534

Purpose

Sets the number of lines per page for the generated source listing.

Syntax



Defaults

The default page size is 66 lines.

Parameters

number
The number of lines per page for the generated source listing.

Usage

The minimum page size that you should set is 25.

IPA effects

This pragma has the same effect on the IPA compile step as it does on a regular compilation. It has no effect on the IPA link step.

#pragma priority (C++ only)

Category

Object code control

Purpose

Specifies the priority level for the initialization of static objects.

The C++ standard requires that all global objects within the same translation unit be constructed from top to bottom, but it does not impose an ordering for objects declared in different translation units. The **#pragma priority** directive allows you to impose a construction order for all static objects declared within the same load module. Destructors for these objects are run in reverse order during termination.

Syntax

Defaults

The default priority level is 0.

Parameters

number
An integer literal in the range of -2147482624 to 2147483647. A lower value indicates a higher priority; a higher value indicates a lower priority. Numbers from -2147483648 to -2147482623 are reserved for system use. If you do not specify a *number*, the compiler assumes 0.

Usage

More than one **#pragma priority** can be specified within a translation unit. The priority value specified in one pragma applies to the constructions of all global objects declared after this pragma and before the next one. However, in order to be consistent with the Standard, priority values specified within the same translation unit must be strictly increasing. Objects with the same priority value are constructed in declaration order.

The effect of a **#pragma priority** exists only within one load module. Therefore, **#pragma priority** cannot be used to control the construction order of objects in different load modules. Refer to *z/OS XL C/C++ Programming Guide* for further discussions on techniques used in handling DLL static object initialization.

#pragma prolog (C only), #pragma epilog (C only)

Category

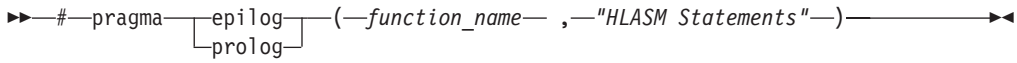
"Object code control" on page 535

Purpose

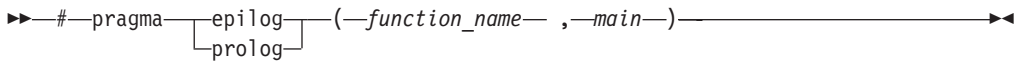
When used with the METAL option, inserts High-Level Assembly (HLASM) prolog or epilog code for a specified function.

Prologs are inserted after function entry, and epilogs are inserted before function return in the generated HLASM code. These directives allow you to provide your own function entry and exit code for system programming.

Syntax



Starting from z/OS V1R11, the following syntax is also supported:



Defaults

Not applicable.

Parameters

function_name
The name of a function to which the epilog or prolog is to be inserted in the generated HLASM code.

HLASM Statements
HLASM Statements is a C string, which must contain valid HLASM statements. If the *HLASM Statements* consists of white-space characters only, or if the *HLASM Statements* is not provided, then the compiler ignores the option specification. If the *HLASM Statements* does not contain any white-space characters, then the compiler will insert leading spaces in front. Otherwise, the

compiler will insert the *HLASM Statements* into the function prolog location of the generated assembler source. The compiler does not understand or validate the contents of the *HLASM Statements*. In order to satisfy the assembly step later, the given *HLASM Statements* must form valid HLASM code with the surrounding code generated by the compiler.

Note: Special characters like `newline` and `quote` are shell (or command line) meta characters, and maybe preprocessed before reaching the compiler. It is advisable to avoid using them.

main
When the keyword *main* is specified instead of *HLASM Statements*, the default prolog/epilog code generated by the compiler for *function_name* is the same as if it was generated for function *main*.

Usage

If a #pragma prolog/epilog directive is specified, the pragma directive overrides its respective compiler option.

An HLASM macro name, or any HLASM statements can be specified in the string parameter. Normal programming language (C/C++) rules for string literals apply to the string arguments of the pragma directives as well as to the compiler options.

These directives are only recognized when the z/OS XL C METAL compiler option is in effect.

Only one **#pragma epilog** or **#pragma prolog** directive is allowed for a specific function.

Related information

- EPILOG, PROLOG, and METAL options in the *z/OS XL C/C++ User's Guide.*
- Default prolog and epilog code information in the *z/OS Metal C Programming Guide and Reference.*

#pragma reachable

Category

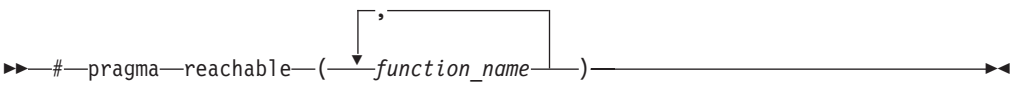
Optimization and tuning

Purpose

Informs the compiler that the point in the program after a named function can be the target of a branch from some unknown location.

By informing the compiler that the instruction after the specified function can be reached from a point in your program other than the return statement in the named function, the pragma allows for additional opportunities for optimization.

Syntax



Parameters

function_name
The name of a function preceding the instruction which is reachable from a point in the program other than the function's return statement.

Defaults

Not applicable.

Usage

Unlike the **#pragma leaves**, **#pragma reachable** is required by the compiler optimizer whenever the instruction following the call may receive control from some program point other than the return statement of the called function. If this condition is true and **#pragma reachable** is not specified, then the subprogram containing the call should not be compiled with OPT(1), OPT(2), OPT(3), or IPA. See also "#pragma leaves" on page 565.

If you specify the LIBANSI compiler option (which informs the compiler that function names that match functions in the C standard library are in fact C library functions), the compiler checks whether the `setjmp` family of functions (`setjmp`, `_setjmp`, `sigsetjmp`, and `_sigsetjmp`) contain **#pragma reachable**. If the functions do not contain this pragma directive, the compiler will insert this directive for the functions. This is not shown in the listing.

IPA effects

If you specify the **#pragma reachable** directive in your source code in the IPA compile step, you cannot override the effects of this directive in the IPA link step.

If you specify the LIBANSI compile option for any translation unit in the IPA compile step, the compiler generates information which indicates the `setjmp` family of functions contain the **reachable** status. If you specify the NOLIBANSI option for the IPA link step, the attribute remains in effect.

Related information

- "#pragma leaves" on page 565

#pragma report (C++ only)

Category

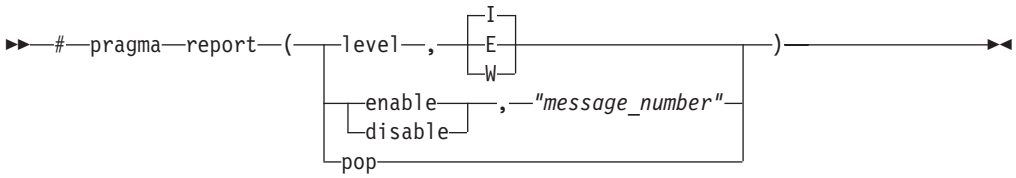
Listings, messages and compiler information

Purpose

Controls the generation of diagnostic messages.

The pragma allows you to specify a minimum severity level for a message for it to display, or allows you to enable or disable a specific message regardless of the prevailing report level.

Syntax



Defaults

The default report level is Informational (**I**), which displays messages of all types.

Parameters

level
Indicates that the pragma is set according to the minimum severity level of diagnostic messages to display.

E Indicates that only error messages will display. Error messages are of the highest severity. This is equivalent to the FLAG(E) compiler option.

W Indicates that warning and error messages will display. This is equivalent to the FLAG(W) compiler option.

I Indicates that all diagnostic messages will display: warning, error and informational messages. Informational messages are of the lowest severity. This is equivalent to the FLAG(I) compiler option.

enable
Enables the specified *"message_number"*.

disable
Disables the specified *"message_number"*.

"message_number"
Represents a message identifier, which consists of a prefix followed by the message number in quotation marks; for example, "CCN1004".

Note: You must use quotation marks with *message_number* as in the preceding example "CCN1004".

pop
Reverts the report level to that which was previously in effect. If no previous report level has been specified, a warning is issued, and the report level remains unchanged.

Usage

The pragma takes precedence over most compiler options. For example, if you use **#pragma report** to disable a compiler message, that message will not be displayed with any FLAG compiler option setting. Similarly, if you specify the SUPPRESS compiler option for a message but also specify **#pragma report(enable)** for the same message, the pragma will prevail.

Related information

- The FLAG option in the *z/OS XL C/C++ User's Guide.*

#pragma runopts

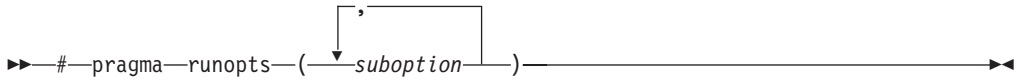
Category

"Language element control" on page 532

Purpose

Specifies a list of runtime options for the compiler to use at execution time.

Syntax



Defaults

Not applicable.

Parameters

suboption
A z/OS Language Environment run-time option or any of the compiler run-time options ARGPARSE, ENV, PLIST, REDIR, or EXECOPS. For more information on z/OS run-time options, see the *z/OS XL C/C++ User's Guide* and *z/OS Language Environment Programming Guide.*

Usage

Specify your **#pragma runopts** directive in the translation unit that contains `main`. If more than one translation unit contains a **#pragma runopts** directive, unpredictable results can occur; the **#pragma runopts** directive only affects translation units containing `main`.

If a suboption to **#pragma runopts** is not a valid C or C++ token, you can surround the suboptions to **#pragma runopts** in double quotes. For example, use:

```
#pragma runopts ( " RPTSTG(ON)
TEST(,,,VADTCPIP&1.2.3.4:*) " )
```

instead of:

```
#pragma runopts ( RPTSTG(ON) TEST(,,,VADTCPIP&1.2.4.3:*))
```

IPA effects

This pragma only affects the IPA compile step if you specify the OBJECT suboption of the IPA compiler option.

The IPA compile step passes the effects of this directive to the IPA link step.

Consider if you specify ARGPARSE | NOARGPARSE, EXECOPS | NOEXECOPS, PLIST, or REDIR | NOREDIR either on the **#pragma runopts** directive or as a compile-time option on the IPA compile step, and then specify the compile-time option on the IPA link step. In this case, you override the value that you specified on the IPA compile step.

If you specify the TARGET compile-time option on the IPA link step, it has the following effects on **#pragma runopts** :

- It overrides the value you specified for **#pragma runopts(ENV)**. If you specify TARGET(LE) or TARGET(), the compiler sets the value of **#pragma runopts(ENV)** to **MVS**. If you specify TARGET(IMS), the compiler sets the value of **#pragma runopts(ENV)** to **IMS**.
- It may override the value you specified for **#pragma runopts(PLIST)**. If you specify TARGET(LE) or TARGET(), and you specified something other than **HOST** for **#pragma runopts(PLIST)**, the compiler sets the value of **#pragma runopts(PLIST)** to **HOST**. If you specify TARGET(IMS), the compiler sets the value of **#pragma runopts(PLIST)** to **IMS**.

For **#pragma runopts** options other than those that are listed above, the IPA link step follows these steps to determine which **#pragma runopts** value to use:

1. The IPA link step uses the **#pragma runopts** specification from the `main` routine, if the routine exists.
2. If no `main` routine exists, the IPA link step follows these steps:
   a. If you define the CEEUOPT variable, the IPA link step uses the **#pragma runopts** value from the first translation unit that it finds that contains CEEUOPT.
   b. If you have not defined the CEEUOPT variable in any translation unit, the IPA link step uses the **#pragma runopts** value from the first translation unit that it processes.

   The sequence of translation unit processing is arbitrary.

To avoid problems, you should specify **#pragma runopts** only in your `main` routine. If you do not have a `main` routine, specify it in only one other module.

Related information

- "#pragma target (C only)" on page 597
- The TARGET option in the *z/OS XL C/C++ User's Guide*.

#pragma sequence

Category

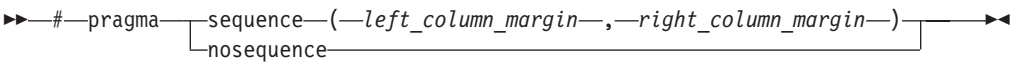
"Language element control" on page 532

Purpose

Defines the section of the input record that is to contain sequence numbers.

The **#pragma nosequence** directive specifies that the input record does not contain sequence numbers.

Syntax



Defaults

- C Sequence numbers are assigned to columns 73 through 80 of the input record for fixed-length-records, and no sequence numbers are assigned for variable-length records. C
- C++ No sequence numbers are assigned for fixed or variable-length records. C++

Parameters

left_column_margin
The column number of the first (left-hand) margin. The value of *left_column_margin* must be greater than 0, and less than 32767.

Also, *left_column_margin* must be less than or equal to the value of *right_column_margin*.

right_column_margin
The column number of the last (right-hand) margin. The value of *right_column_margin* must be greater than that of *left_column_margin*, and less than 32767.

You can also use an asterisk (*) to indicate the last column of the input record. For example, **sequence(74,*)** indicates that sequence numbers are between column 74 and the end of the input record.

Usage

C++ In C++ only, you can specify the **sequence** option with no parameters, which instructs the compiler to number columns 73 through 80 of the input record (fixed or variable length). C++

If you use the compiler options SEQUENCE or NOSEQUENCE with the **#pragma sequence/nosequence** directive, the directive overrides the compiler options. The compiler option is in effect up to the first **#pragma sequence/nosequence** directive. The sequence setting specified by the **#pragma sequence** directive applies only to the file (source file or include file) that contains it. The setting has no effect on other include files in the file.

You can use **#pragma sequence** and **#pragma margins** together. If they reserve the same columns, **#pragma sequence** has priority, and the compiler reserves the columns for sequence numbers. For example, consider if the columns reserved for the margin are 1 to 20 and the columns reserved for sequence numbers are 15 to 25. In this case, the margin will be from column 1 to 14, and the columns reserved for sequence numbers will be from 15 to 25. For more information on the **#pragma margins** directive, refer to "#pragma margins/nomargins" on page 572.

Related information

- The SEQUENCE compiler option in the *z/OS XL C/C++ User's Guide.*

#pragma skip (C only)

Category

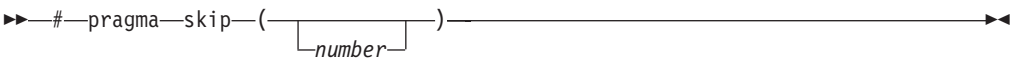
"Listings, messages and compiler information" on page 534

Purpose

Skips lines of the generated source listing.

Syntax

#pragma skip directive syntax



Defaults

Not applicable.

Parameters

number
The number of lines to skip in the generated source listing. The value of *number* must be a positive integer less than 255. If you omit *number*, the compiler skips one line.

#pragma strings

Category

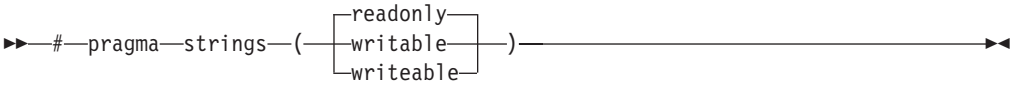
Object code control

Purpose

Specifies the storage type for string literals.

Syntax

Pragma syntax



Defaults

C and C++ strings are read-only.

Parameters

readonly
String literals are to be placed in read-only memory.

writeable
String literals are to be placed in read-write memory.

Usage

Placing string literals in read-only memory can improve runtime performance and save storage. However, code that attempts to modify a read-only string literal may generate a memory error.

The pragma must appear before any source statements in a file.

IPA effects

During the IPA link step, the compiler compares the **#pragma strings** specifications for individual translation units. If it finds differences, it treats the strings as if you specified **#pragma strings(writeable)** for all translation units.

Related information

- The ROSTRINGS option in the *z/OS XL C/C++ User's Guide.*

#pragma subtitle (C only)

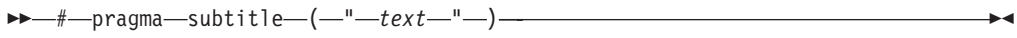
Category

"Listings, messages and compiler information" on page 534

Purpose

Places subtitle text on all subsequent pages of the generated source listing.

Syntax



Defaults

Not applicable.

Parameters

text
The subtitle to appear in on all pages of the generated source listing.

#pragma target (C only)

Category

"Object code control" on page 535

Purpose

Specifies the operating system or runtime environment for which the compiler creates the object module.

Syntax

Defaults

The default is **LE**.

Parameters

LE Generates code to run under the z/OS Language Environment run-time library. This is equivalent to specifying **#pragma target()**. This suboption has the following effects on **#pragma runopts(ENV)** and **#pragma runopts(PLIST)**:

- If you did not specify values for **#pragma runopts(ENV)** or **#pragma runopts(PLIST)**, the compiler sets the pragmas to **#pragma runopts(ENV(MVS))** and **#pragma runopts(PLIST(HOST))**.
- If you did specify values for **#pragma runopts(ENV)** or **#pragma runopts(PLIST)**, the values do not change.

IMS
Generates object code to run under IMS. This suboption has the following effects on **#pragma runopts(ENV)** and **#pragma runopts(PLIST)** :

- If you did not specify values for **#pragma runopts(ENV)** or **#pragma runopts(PLIST)**, the compiler sets the pragmas to **#pragma runopts(ENV(IMS))** and **#pragma runopts(PLIST(OS))**.
- If you did specify values for **#pragma runopts(ENV)** or **#pragma runopts(PLIST)**, the values do not change.

Usage

Note that you cannot specify the release suboptions using the **#pragma target** directive as you can with the TARGET compiler option.

The only pragma directives that can precede **#pragma target** are **filetag**, **chars**, **langlvl**, and **longname**.

IPA effects

This pragma only affects the IPA compile step if you specify the OBJECT suboption of the IPA compiler option.

The IPA compile step passes the effects of this pragma to the IPA link step.

If you specify different **#pragma target** directives for different translation units, the IPA link step uses the ENV and PLIST information from the translation unit containing `main`. If there is no `main`, it uses information from the first translation unit it finds. If you specify the TARGET compile option for the IPA link step, it overrules the **#pragma target** directive.

Related information

- "#pragma runopts" on page 593
- The TARGET option in the *z/OS XL C/C++ User's Guide*.

#pragma title (C only)

Category

"Listings, messages and compiler information" on page 534

Purpose

Places title text on all subsequent pages of the generated source listing.

Syntax

Defaults

Not applicable.

Parameters

text
The title to appear in on all pages of the generated source listing.

#pragma unroll

Category

Optimization and tuning

Purpose

Controls loop unrolling, for improved performance.

When **unroll** is in effect, the optimizer determines and applies the best unrolling factor for each loop; in some cases, the loop control might be modified to avoid unnecessary branching. The compiler remains the final arbiter of whether the loop is unrolled.

Syntax

Pragma syntax

Defaults

See the description of the UNROLL option in the *z/OS XL C/C++ User's Guide.*

Parameters

The following suboptions are for **-qunroll** only.

n Instructs the compiler to unroll loops by a factor of *n*. In other words, the body of a loop is replicated to create *n* copies and the number of iterations is reduced by a factor of *1/n*. The **-qunroll=n** option specifies a global unroll factor that affects all loops that do not have an unroll pragma yet. The value of *n* must be a positive integer.

Specifying **#pragma unroll(1)** or **-qunroll=1** disables loop unrolling, and is equivalent to specifying **#pragma nounroll** or **-qnounroll**.

The compiler might limit unrolling to a number smaller than the value you specify for *n*. This is because the option form affects all loops in source files to which it applies and large unrolling factors might significantly increase

compile time without necessarily improving runtime performance. To specify an unrolling factor for particular loops, use the #pragma form in those loops.

Usage

The pragma overrides the [NO]UNROLL option setting for a designated loop. However, even if **#pragma unroll** is specified for a given loop, the compiler remains the final arbiter of whether the loop is unrolled.

Only one pragma can be specified on a loop.

The pragma affects only the loop that follows it. An inner nested loop requires a **#pragma unroll** directive to precede it if the wanted loop unrolling strategy is different from that of the prevailing [NO]UNROLL option.

The **#pragma unroll** and **#pragma nounroll** directives can only be used on `for` loops. They cannot be applied to `do while` and `while` loops.

The loop structure must meet the following conditions:

- There must be only one loop counter variable, one increment point for that variable, and one termination variable. These cannot be altered at any point in the loop nest.
- Loops cannot have multiple entry and exit points. The loop termination must be the only means to exit the loop.
- Dependencies in the loop must not be "backwards-looking". For example, a statement such as `A[i][j] = A[i -1][j + 1] + 4` must not appear within the loop.

Predefined macros

None.

Examples

In the following example, the **#pragma unroll(3)** directive on the first `for` loop requires the compiler to replicate the body of the loop three times. The **#pragma unroll** on the second `for` loop allows the compiler to decide whether to perform unrolling.

```
#pragma unroll(3)
for( i=0;i < n; i++)
{
      a[i] = b[i] * c[i];
}

#pragma unroll
for( j=0;j < n; j++)
{
      a[j] = b[j] * c[j];

}
```

In this example, the first **#pragma unroll(3)** directive results in:

```
i=0;
if (i>n-2) goto remainder;
for (; i<n-2; i+=3) {
  a[i]=b[i] * c[i];
  a[i+1]=b[i+1] * c[i+1];
  a[i+2]=b[i+2] * c[i+2];
```

```
}
if (i<n) {
  remainder:
  for (; i<n; i++) {
    a[i]=b[i] * c[i];
  }
}
```

Related information

- The UNROLL option in the *z/OS XL C/C++ User's Guide*.

#pragma variable

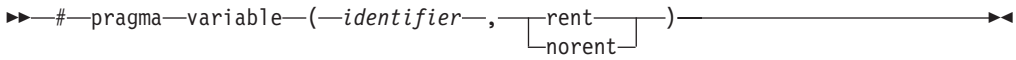
Category

"Object code control" on page 535

Purpose

Specifies whether the compiler is to use a named external object in a reentrant or non-reentrant fashion.

Syntax



Defaults

C++ Variables are reentrant (**rent**). C++

C Variables are not reentrant (**norent**). C

Parameters

identifier
The name of an external variable.

rent
Specifies that the variable's references or definition will be in the writable static area that is in modifiable storage.

norent
Specifies that the variable's references or its definition is in the code area and is in potentially read-only storage. This suboption does not apply to, and has no effect on, program variables with `static` storage class.

Usage

If an identifier is defined in one translation unit and used in another, the reentrant or non-reentrant status of the variable must be the same in all translation units.

C To specify that variables declared as `const` not be placed into the writeable static area, you must use the ROCONST and RENT compiler options. C

If the specification for a `const` variable in a **#pragma variable** directive conflicts with the ROCONST option, the pragma directive takes precedence over the compiler option, and the compiler issues an informational message.

If you use the **norent** suboption for a variable, ensure that your program never writes to this variable. Program exceptions or unpredictable program behavior may result should this be the case.

The following code fragment leads to undefined behavior when compiled with the RENT option.

```
int i;
int *p = &i;
#pragma variable(p, norent)
```

The variable `i` is reentrant, but the pointer `p` is non-reentrant. If the code is in a DLL, there will only be one copy of `p` but multiple copies of `i`, one for each caller of the DLL.

A non-reentrant pointer variable cannot take an address as an initializer: the compiler will treat the variable as reentrant if necessary (in other words, it will ignore the pragma). Initializers for non-reentrant variables should be compile-time constants. Due to code relocation during execution time, an address in a program that has both reentrant and non-reentrant variables is never considered a compile-time constant. This restriction includes the addresses of string literals.

Examples

The following program fragment shows how to use the **#pragma variable** directive to force an external program variable to be part of a program that includes executable code and constant data.

```
#pragma variable(rates, norent)
extern float rates[5] = { 3.2, 83.3, 13.4, 3.6, 5.0 };

extern float totals[5];

int main(void) {
⋮
}
```

In this example, you compile the source file with the RENT option. The executable code includes the variable **rates** because **#pragma variable(rates, norent)** is specified. The writeable static area includes the variable **totals**. Each user has a personal copy of the array **totals**, and all users of the program share the array **rates**. This sharing may yield a performance and storage benefit.

Related information

- The RENT option in the *z/OS XL C/C++ User's Guide.*

#pragma wsizeof

Category

"Portability and migration" on page 536

Purpose

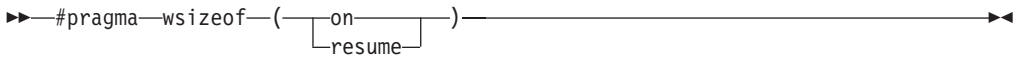
Toggles the behavior of the `sizeof` operator between that of a compiler prior the C/C++ for MVS/ESA V3R2 and the z/OS XL C/C++ compiler.

When the `sizeof` operator was applied to a function return type, older z/OS C and C++ compilers returned the size of the widened type instead of the original type. For example, in the following code fragment, using the older compilers, `i` has a value of 4.

```
char foo();
i = sizeof foo();
```

Using the z/OS XL C/C++ compiler, `i` has a the value of 1, which is the size of the original type, `char`.

Syntax



Defaults

The `sizeof` operator returns the original type for function return types.

Parameters

on Enables the old compiler behavior of the `sizeof` operator, so that the widened size is returned for function return types.

resume
Re-enables the normal behavior of the `sizeof` operator.

Usage

You can use this pragma in old header files where you require the old behavior of the `sizeof` operator. By guarding the header file with a **#pragma wsizeof(on)** at the start of the header, and a **#pragma wsizeof(resume)** at the end, you can use the old header file with new applications.

The compiler will match **on** and **resume** throughout the entire compilation unit. That is, the effect of the pragma can extend beyond a header file. Ensure the **on** and **resume** pragmas are matched in your compilation unit.

The pragma only affects the use of `sizeof` on function return types. Other behaviors of `sizeof` remain the same.

Note: Dangling the **resume** pragma leads to undefined behavior. The effect of an unmatched **on** pragma can extend to the end of the source file.

IPA effects

During the IPA compile step, the size of each function return value is resolved during source processing. The IPA compile and link steps do not alter these sizes. The IPA object code from translation units with different **wsizeof** settings is merged together during the IPA link step.

Related information

- The WSIZEOF option in the *z/OS XL C/C++ User's Guide.*

#pragma XOPTS

Category

"Language element control" on page 532

Purpose

Passes suboptions directly to the CICS integrated translator for processing CICS statements embedded in C/C++ source code.

Syntax

Defaults

Not applicable.

Parameters

XOPTS
Must be specified in all uppercase.

suboptions
Are options to be passed to the CICS integrated translator.

Usage

The directive is only valid when the CICS compiler option is in effect. It must appear before any C/C++ or CICS statements in the source, and must appear at file scope (C) or global namespace scope (C++).

Note that if you invoke the compiler with any of the preprocessing-only options, the directive will be preserved in the preprocessed output.

Related information

For detailed information on acceptable embedded CICS statements and preprocessed output, see the description of the CICS compiler option in the *z/OS XL C/C++ User's Guide.*

Pragma directives for parallel processing

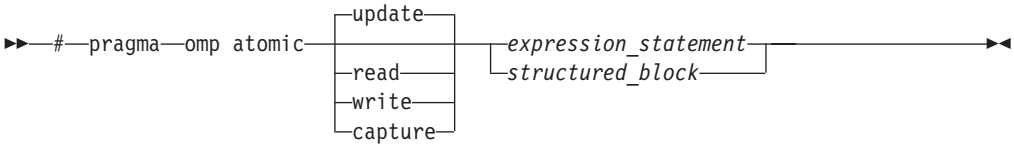
Parallel processing operations are controlled by pragma directives in your program source. The pragmas have effect only when parallelization is enabled with the **SMP** compiler option.

You can use the OpenMP directives in your C and C++ programs. Each directive has its own usage characteristics.

#pragma omp atomic
Purpose

The **omp atomic** directive allows access of a specific memory location atomically. It ensures that race conditions are avoided through direct control of concurrent threads that might read or write to or from the particular memory location. With the **omp atomic** directive, you can write more efficient concurrent algorithms with fewer locks.

Syntax



where *expression_statement* is an expression statement of scalar type, and *structured_block* is a structured block of two expression statements.

Clauses

update
Updates the value of a variable atomically. Guarantees that only one thread at a time updates the shared variable, avoiding errors from simultaneous writes to the same variable. An **omp atomic** directive without a clause is equivalent to an **omp atomic** update.

Note: Atomic updates cannot write arbitrary data to the memory location, but depend on the previous data at the memory location.

read
Reads the value of a variable atomically. The value of a shared variable can be read safely, avoiding the danger of reading an intermediate value of the variable when it is accessed simultaneously by a concurrent thread.

write
Writes the value of a variable atomically. The value of a shared variable can be written exclusively to avoid errors from simultaneous writes.

capture
Updates the value of a variable while capturing the original or final value of the variable atomically.

The *expression_statement* or *structured_block* takes one of the following forms, depending on the atomic directive clause:

Directive clause	*expression_statement*	*structured_block*

update
(equivalent to no clause)
	`x++;`
	`x--;`
	`++x;`
	`--x;`
	`x binop = expr;`
	`x = x binop expr;`
	`x = expr binop x;`	

read	`v = x;`	
write	`x = expr;`	

capture	`v = x++;`
	`v = x--;`
	`v = ++x;`
	`v = --x;`
	`v = x binop = expr;`
	`v = x = x binop expr;`
	`v = x = expr binop x;`
		`{v = x; x binop = expr;}`
		`{v = x; xOP;}`
		`{v = x; OPx;}`
		`{x binop = expr; v = x;}`
		`{xOP; v = x;}`
		`{OPx; v = x;}`
		`{v = x; x = x binop expr;}`
		`{x = x binop expr; v = x;}`
		`{v = x; x = expr binop x;}`
		`{x = expr binop x; v = x;}`
		`{v = x; x = expr;}`[1]

Note:
1. This expression is to support atomic swap operations.

where:

x, v are both lvalue expressions with scalar type.

expr is an expression of scalar type that does not reference *x*.

binop is one of the following binary operators:
`+ * - / & ^ | << >>`

OP is one of `++` or `--`.

Note: *binop*, *binop*=, and *OP* are not overloaded operators.

Usage

Objects that can be updated in parallel and that might be subject to race conditions should be protected with the **omp atomic** directive.

All atomic accesses to the storage locations designated by *x* throughout the program should have a compatible type.

Within an atomic region, multiple syntactic occurrences of *x* must designate the same storage location.

All accesses to a certain storage location throughout a concurrent program must be atomic. A non-atomic access to a memory location might break the expected atomic behavior of all atomic accesses to that storage location.

Neither *v* nor *expr* can access the storage location that is designated by *x*.

Neither *x* nor *expr* can access the storage location that is designated by *v*.

All accesses to the storage location designated by *x* are atomic. Evaluations of the expression *expr*, *v*, *x* are not atomic.

For atomic capture access, the operation of writing the captured value to the storage location represented by *v* is not atomic.

Examples

Example 1: Atomic update

```
extern float x[], *p = x, y;

/* Protect against race conditions among multiple updates. */
#pragma omp atomic
x[index[i]] += y;

/* Protect against race conditions with updates through x. */
#pragma omp atomic
p[i] -= 1.0f;
```

Example 2: Atomic read, write, and update

```
extern int x[10];
extern int f(int);
int temp[10], i;

for(i = 0; i < 10; i++)
{
  #pragma omp atomic read
  temp[i] = x[f(i)];

  #pragma omp atomic write
  x[i] = temp[i]*2;

  #pragma omp atomic update
  x[i] *= 2;
}
```

Example 3: Atomic capture

```
extern int x[10];
extern int f(int);
int temp[10], i;

for(i = 0; i < 10; i++)
{
  #pragma omp atomic capture
  temp[i] = x[f(i)]++;

  #pragma omp atomic capture
  {
```

```
    temp[i] = x[f(i)]; //the two occurences of x[f(i)] must evaluate to the
    x[f(i)] -= 3; //same memory location, otherwise behavior is undefined.
  }
}
```

#pragma omp barrier
Purpose

The **omp barrier** directive identifies a synchronization point at which threads in a parallel region will not execute beyond the **omp barrier** until all other threads in the team complete all explicit tasks in the region.

Syntax

Usage

The **omp barrier** directive must appear within a block or compound statement. For example:

```
if (x!=0) {
   #pragma omp barrier    /* valid usage    */
}
if (x!=0)
   #pragma omp barrier    /* invalid usage  */
```

#pragma omp critical
Purpose

The **omp critical** directive identifies a section of code that must be executed by a single thread at a time.

Syntax

where *name* can optionally be used to identify the critical region. Identifiers naming a critical region have external linkage and occupy a namespace distinct from that used by ordinary identifiers.

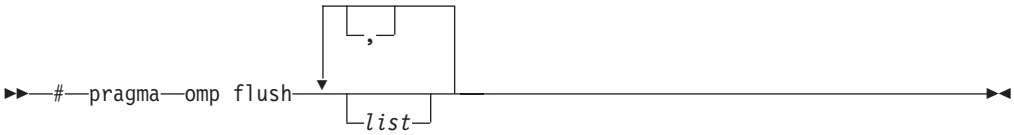
Usage

A thread waits at the start of a critical region identified by a given name until no other thread in the program is executing a critical region with that same name. Critical sections not specifically named by **omp critical** directive invocation are mapped to the same unspecified name.

#pragma omp flush
Purpose

The **omp flush** directive identifies a point at which the compiler ensures that all threads in a parallel region have the same view of specified objects in memory.

Syntax



where *list* is a comma-separated list of variables that will be synchronized.

Usage

If *list* includes a pointer, the pointer is flushed, not the object being referred to by the pointer. If *list* is not specified, all shared objects are synchronized except those inaccessible with automatic storage duration.

An implied **flush** directive appears in conjunction with the following directives:

- **omp barrier**
- Entry to and exit from **omp critical**.
- Exit from **omp parallel**.
- Exit from **omp for**.
- Exit from **omp sections**.
- Exit from **omp single**.

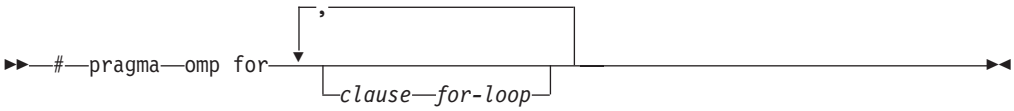
The **omp flush** directive must appear within a block or compound statement. For example:

```
if (x!=0) {
   #pragma omp flush    /* valid usage    */
}
if (x!=0)
   #pragma omp flush    /* invalid usage  */
```

#pragma omp for
Purpose

The **omp for** directive instructs the compiler to distribute loop iterations within the team of threads that encounters this work-sharing construct.

Syntax



Parameters

clause is any of the following clauses:

collapse (*n*)
Allows you to parallelize multiple loops in a nest without introducing nested parallelism.

- Only one collapse clause is allowed on a worksharing **for** or **parallel for** pragma.
- The specified number of loops must be present lexically. That is, none of the loops can be in a called subroutine.
- The loops must form a rectangular iteration space and the bounds and stride of each loop must be invariant over all the loops.
- If the loop indices are of different size, the index with the largest size will be used for the collapsed loop.
- The loops must be perfectly nested; that is, there is no intervening code nor any OpenMP pragma between the loops which are collapsed.
- The associated do-loops must be structured blocks. Their execution must not be terminated by an **break** statement.
- If multiple loops are associated to the loop construct, only an iteration of the innermost associated loop may be curtailed by a **continue** statement. If multiple loops are associated to the loop construct, there must be no branches to any of the loop termination statements except for the innermost associated loop.

Ordered construct
During execution of an iteration of a loop or a loop nest within a loop region, the executing thread must not execute more than one ordered region which binds to the same loop region. As a consequence, if multiple loops are associated to the loop construct by a collapse clause, the ordered construct has to be located inside all associated loops.

Lastprivate clause
When a lastprivate clause appears on the pragma that identifies a work-sharing construct, the value of each new list item from the sequentially last iteration of the associated loops, is assigned to the original list item even if a collapse clause is associated with the loop

Other SMP and performance pragmas
stream_unroll,**unroll**,**unrollandfuse**,**nounrollandfuse** pragmas cannot be used for any of the loops associated with the **collapse** clause loop nest.

private (*list***)**
Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.

firstprivate (*list***)**
Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.

lastprivate (*list***)**
Declares the scope of the data variables in *list* to be private to each thread. The final value of each variable in *list*, if assigned, will be the value assigned to that variable in the last iteration. Variables not assigned a value will have an indeterminate value. Data variables in *list* are separated by commas.

reduction (*operator***:** *list***)**
Performs a reduction on all scalar variables in *list* using the specified *operator*. Reduction variables in *list* are separated by commas.

A private copy of each variable in *list* is created for each thread. At the end of the statement block, the final values of all private copies of the reduction variable are combined in a manner appropriate to the operator, and the result is placed back in the original value of the shared reduction variable. For example, when the `max` operator is specified, the original reduction variable value combines with the final values of the private copies by using the following expression:

```
original_reduction_variable = original_reduction_variable < private_copy ?
private_copy : original_reduction_variable;
```

For variables specified in the **reduction** clause, they must satisfy the following conditions:

- Must be of a type appropriate to the operator. If the `max` or `min` operator is specified, the variables must be one of the following types with or without `long`, `short`, `signed`, or `unsigned`:
  - `_Bool` (C only)
  - `bool` (C++ only)
  - `char`
  - `wchar_t` (C++ only)
  - `int`
  - `float`
  - `double`
- Must be shared in the enclosing context.
- Must not be const-qualified.
- Must not have pointer type.

ordered
Specify this clause if an ordered construct is present within the dynamic extent of the **omp for** directive.

schedule (*type***)**
Specifies how iterations of the **for** loop are divided among available threads. Acceptable values for *type* are:

auto With **auto**, scheduling is delegated to the compiler and runtime system. The compiler and runtime system can choose any possible mapping of iterations to threads (including all possible valid schedules) and these may be different in different loops.

dynamic
Iterations of a loop are divided into chunks of size **ceiling**(*number_of_iterations*/*number_of_threads*).

Chunks are dynamically assigned to active threads on a "first-come, first-do" basis until all work has been assigned.

dynamic,*n*
As above, except chunks are set to size *n*. *n* must be an integral assignment expression of value 1 or greater.

guided
Chunks are made progressively smaller until the default minimum chunk size is reached. The first chunk is of size **ceiling**(*number_of_iterations*/*number_of_threads*). Remaining chunks are of size **ceiling**(*number_of_iterations_left*/*number_of_threads*).

The minimum chunk size is 1.

Chunks are assigned to active threads on a "first-come, first-do" basis until all work has been assigned.

guided,*n*
As above, except the minimum chunk size is set to *n*; *n* must be an integral assignment expression of value 1 or greater.

runtime
Scheduling policy is determined at run time. Use the OMP_SCHEDULE environment variable to set the scheduling type and chunk size.

static Iterations of a loop are divided into chunks of size **ceiling**(*number_of_iterations*/*number_of_threads*). Each thread is assigned a separate chunk.

This scheduling policy is also known as *block scheduling*.

static,*n*
Iterations of a loop are divided into chunks of size *n*. Each chunk is assigned to a thread in *round-robin* fashion.

n must be an integral assignment expression of value 1 or greater.

This scheduling policy is also known as *block cyclic scheduling*.

nowait
Use this clause to avoid the implied **barrier** at the end of the **for** directive. This is useful if you have multiple independent work-sharing sections or iterative loops within a given parallel region. Only one **nowait** clause can appear on a given **for** directive.

and where *for_loop* is a `for` loop construct with the following canonical shape:

```
for (init_expr; exit_cond; incr_expr)
 statement
```

where:

init_expr takes the form:
```
iv = b
integer-type iv = b
```

exit_cond takes the form:
```
iv <= ub
iv <  ub
iv >= ub
iv >  ub
```

incr_expr takes the form:
```
++iv
iv++
--iv
iv--
iv += incr
iv -= incr
iv = iv + incr
iv = incr + iv
iv = iv - incr
```

and where:

iv Iteration variable. The iteration variable must be a `signed integer` not modified anywhere within the `for` loop. It is implicitly made private for the duration of the `for` operation. If not specified as **lastprivate**, the iteration variable will have an indeterminate value after the operation completes.

b, *ub*, *incr* Loop invariant signed integer expressions. No synchronization is performed when evaluating these expressions and evaluated side effects may result in indeterminate values.

Usage

This pragma must appear immediately before the loop or loop block directive to be affected.

Program sections using the **omp for** pragma must be able to produce a correct result regardless of which thread executes a particular iteration. Similarly, program correctness must not rely on using a particular scheduling algorithm.

The `for` loop iteration variable is implicitly made private in scope for the duration of loop execution. This variable must not be modified within the body of the `for` loop. The value of the increment variable is indeterminate unless the variable is specified as having a data scope of **lastprivate**.

An implicit barrier exists at the end of the `for` loop unless the **nowait** clause is specified.

Restrictions:

- The `for` loop must be a structured block, and must not be terminated by a `break` statement.
- Values of the loop control expressions must be the same for all iterations of the loop.
- An **omp for** directive can accept only one **schedule** clause.
- The value of *n* (chunk size) must be the same for all threads of a parallel region.

#pragma omp master
Purpose

The **omp master** directive identifies a section of code that must be run only by the master thread.

Syntax

Usage

Threads other than the master thread will not execute the statement block associated with this construct.

No implied barrier exists on either entry to or exit from the master section.

#pragma omp ordered
Purpose

The **omp ordered** directive identifies a structured block of code that must be executed in sequential order.

Syntax

Usage

The **omp ordered** directive must be used as follows:

- It must appear within the extent of a **omp for** or **omp parallel for** construct containing an **ordered** clause.
- It applies to the statement block immediately following it. Statements in that block are executed in the same order in which iterations are executed in a sequential loop.
- An iteration of a loop must not execute the same **omp ordered** directive more than once.
- An iteration of a loop must not execute more than one distinct **omp ordered** directive.

#pragma omp parallel
Purpose

The **omp parallel** directive explicitly instructs the compiler to parallelize the chosen block of code.

Syntax

Parameters

clause is any of the following clauses:

if (*exp***)**
When the `if` argument is specified, the program code executes in parallel only if the scalar expression represented by *exp* evaluates to a nonzero value at run time. Only one `if` clause can be specified.

private (*list***)**
Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.

firstprivate (*list***)**
Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized with the value of the original variable as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.

num_threads (*int_exp***)**
The value of *int_exp* is an integer expression that specifies the number of threads to use for the parallel region. If dynamic adjustment of the number of threads is also enabled, then *int_exp* specifies the maximum number of threads to be used.

shared (*list***)**
Declares the scope of the comma-separated data variables in *list* to be shared across all threads.

default (shared | none)
Defines the default data scope of variables in each thread. Only one **default** clause can be specified on an **omp parallel** directive.

Specifying **default(shared)** is equivalent to stating each variable in a **shared(***list***)** clause.

Specifying **default(none)** requires that each data variable visible to the parallelized statement block must be explcitly listed in a data scope clause, with the exception of those variables that are:
- const-qualified,
- specified in an enclosed data scope attribute clause, or,
- used as a loop control variable referenced only by a corresponding **omp for** or **omp parallel for** directive.

copyin (*list***)**
For each data variable specified in *list*, the value of the data variable in the master thread is copied to the thread-private copies at the beginning of the parallel region. Data variables in *list* are separated by commas.

Each data variable specified in the **copyin** clause must be a **threadprivate** variable.

reduction (*operator***: *list*)**
Performs a reduction on all scalar variables in *list* using the specified *operator*. Reduction variables in *list* are separated by commas.

A private copy of each variable in *list* is created for each thread. At the end of the statement block, the final values of all private copies of the reduction variable are combined in a manner appropriate to the operator, and the result is placed back in the original value of the shared reduction variable. For example, when the `max` operator is specified, the original reduction variable value combines with the final values of the private copies by using the following expression:

```
original_reduction_variable = original_reduction_variable < private_copy ?
private_copy : original_reduction_variable;
```

For variables specified in the **reduction** clause, they must satisfy the following conditions:
- Must be of a type appropriate to the operator. If the `max` or `min` operator is specified, the variables must be one of the following types with or without `long`, `short`, `signed`, or `unsigned`:
  - `_Bool` (C only)
  - `bool` (C++ only)
  - `char`
  - `wchar_t` (C++ only)
  - `int`
  - `float`
  - `double`
- Must be shared in the enclosing context.
- Must not be const-qualified.
- Must not have pointer type.

Usage

When a parallel region is encountered, a logical team of threads is formed. Each thread in the team executes all statements within a parallel region except for work-sharing constructs. Work within work-sharing constructs is distributed among the threads in a team.

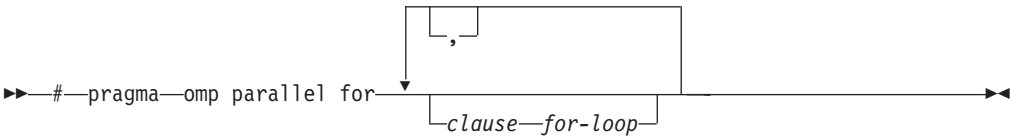
Loop iterations must be independent before the loop can be parallelized. An implied barrier exists at the end of a parallelized statement block.

By default, nested parallel regions are serialized.

#pragma omp parallel for
Purpose

The **omp parallel for** directive effectively combines the **omp parallel** and **omp for** directives. This directive lets you define a parallel region containing a single **for** directive in one step.

Syntax



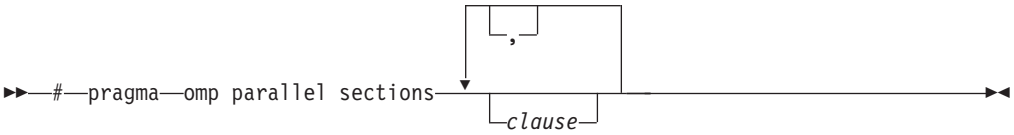
Usage

With the exception of the **nowait** clause, clauses and restrictions described in the **omp parallel** and **omp for** directives also apply to the **omp parallel for** directive.

#pragma omp parallel sections
Purpose

The **omp parallel sections** directive effectively combines the **omp parallel** and **omp sections** directives. This directive lets you define a parallel region containing a single **sections** directive in one step.

Syntax



Usage

All clauses and restrictions described in the **omp parallel** and **omp sections** directives apply to the **omp parallel sections** directive.

#pragma omp section, #pragma omp sections
Purpose

The **omp sections** directive distributes work among threads bound to a defined parallel region.

Syntax

Parameters

clause is any of the following clauses:

private (*list*)
Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.

firstprivate (*list*)
Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.

lastprivate (*list*)
Declares the scope of the data variables in *list* to be private to each thread. The final value of each variable in *list*, if assigned, will be the value assigned to that variable in the last **section**. Variables not assigned a value will have an indeterminate value. Data variables in *list* are separated by commas.

reduction (*operator*: *list*)
Performs a reduction on all scalar variables in *list* using the specified *operator*. Reduction variables in *list* are separated by commas.

A private copy of each variable in *list* is created for each thread. At the end of the statement block, the final values of all private copies of the reduction variable are combined in a manner appropriate to the operator, and the result is placed back in the original value of the shared reduction variable. For example, when the `max` operator is specified, the original reduction variable value combines with the final values of the private copies by using the following expression:

```
original_reduction_variable = original_reduction_variable < private_copy ?
private_copy : original_reduction_variable;
```

For variables specified in the **reduction** clause, they must satisfy the following conditions:

- Must be of a type appropriate to the operator. If the `max` or `min` operator is specified, the variables must be one of the following types with or without `long`, `short`, `signed`, or `unsigned`:
  - `_Bool` (C only)
  - `bool` (C++ only)
  - `char`
  - `wchar_t` (C++ only)
  - `int`
  - `float`
  - `double`
- Must be shared in the enclosing context.
- Must not be const-qualified.
- Must not have pointer type.

nowait
Use this clause to avoid the implied **barrier** at the end of the **sections** directive. This is useful if you have multiple independent work-sharing sections within a given parallel region. Only one **nowait** clause can appear on a given **sections** directive.

Usage

The **omp section** directive is optional for the first program code segment inside the **omp sections** directive. Following segments must be preceded by an **omp section**

directive. All **omp section** directives must appear within the lexical construct of the program source code segment associated with the **omp sections** directive.

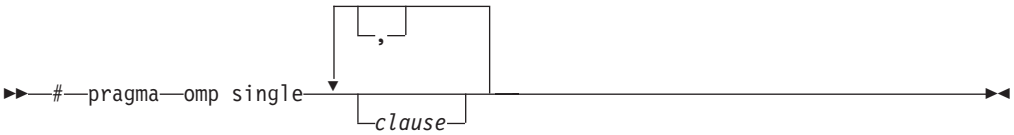
When program execution reaches a **omp sections** directive, program segments defined by the following **omp section** directive are distributed for parallel execution among available threads. A barrier is implicitly defined at the end of the larger program region associated with the **omp sections** directive unless the **nowait** clause is specified.

#pragma omp single

Purpose

The **omp single** directive identifies a section of code that must be run by a single available thread.

Syntax



Parameters

clause is any of the following:

private (*list*)
Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.

A variable in the **private** clause must not also appear in a **copyprivate** clause for the same **omp single** directive.

copyprivate (*list*)
Broadcasts the values of variables specified in *list* from one member of the team to other members. This occurs after the execution of the structured block associated with the **omp single** directive, and before any of the threads leave the barrier at the end of the construct. For all other threads in the team, each variable in the *list* becomes defined with the value of the corresponding variable in the thread that executed the structured block. Data variables in *list* are separated by commas. Usage restrictions for this clause are:

- A variable in the **copyprivate** clause must not also appear in a **private** or **firstprivate** clause for the same **omp single** directive.
- If an **omp single** directive with a **copyprivate** clause is encountered in the dynamic extent of a parallel region, all variables specified in the **copyprivate** clause must be private in the enclosing context.
- Variables specified in **copyprivate** clause within dynamic extent of a parallel region must be private in the enclosing context.
- A variable that is specified in the **copyprivate** clause must have an accessible and unambiguous copy assignment operator.
- The **copyprivate** clause must not be used together with the **nowait** clause.

firstprivate (*list*)
Declares the scope of the data variables in *list* to be private to each thread.

Each new private object is initialized as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.

A variable in the **firstprivate** clause must not also appear in a **copyprivate** clause for the same **omp single** directive.

nowait
Use this clause to avoid the implied **barrier** at the end of the **single** directive. Only one **nowait** clause can appear on a given **single** directive. The **nowait** clause must not be used together with the **copyprivate** clause.

Usage

An implied barrier exists at the end of a parallelized statement block unless the **nowait** clause is specified.

#pragma omp task
Purpose

The **task** pragma can be used to explicitly define a task.

Use the **task** pragma when you want to identify a block of code to be executed in parallel with the code outside the task region. The **task** pragma can be useful for parallelizing irregular algorithms such as pointer chasing or recursive algorithms. The **task** directive takes effect only if you specify the **SMP** compiler option.

Syntax

Parameters

The *clause* parameter can be any of the following types of clauses:

default (shared | none)
Defines the default data scope of variable in each task. Only one `default` clause can be specified on an `omp task` directive.

Specifying `default(shared)` is equivalent to stating each variable in a `shared(list)` clause.

Specifying `default(none)` requires that each data variable visible to the construct must be explicitly listed in a data scope clause, with the exception of variables with the following attributes:

- Threadprivate
- Automatic and declared in a scope inside the construct
- Objects with dynamic storage duration
- Static data members
- The loop iteration variables in the associated for-loops for a work-sharing **for** or **parallel for** construct
- Static and declared in a scope inside the construct

final (*exp*)
If you specify a `final` clause and *exp* evaluates to a nonzero value, the generated task is a final task. All task constructs encountered inside a final task create final and included tasks.

You can specify only one `final` clause on the **task** pragma.

firstprivate (*list*)
Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized with the value of the original variable as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.

if (*exp*)
When the `if` clause is specified, an undeferred task is generated if the scalar expression *exp* evaluates to a nonzero value. Only one `if` clause can be specified.

mergeable
If you specify a `mergeable` clause and the generated task is an undeferred task or included task, a merged task might be generated.

private (*list*)
Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.

shared (*list*)
Declares the scope of the comma-separated data variables in *list* to be shared across all threads.

untied
When a task region is suspended, untied tasks can be resumed by any thread in a team. The `untied` clause on a task construct is ignored if either of the following conditions is met:

- A `final` clause is specified on the same task construct and the `final` clause expression evaluates to a nonzero value.
- The task is an included task.

Usage

A final task is a task that makes all its child tasks become final and included tasks. A final task is generated when either of the following conditions is a nonzero value:

- A `final` clause is specified on a task construct and the `final` clause expression evaluates to nonzero value.
- The generated task is a child task of a final task.

An undeferred task is a task whose execution is not deferred with respect to its generating task region. In other words, the generating task region is suspended until the undeferred task has finished running. An undeferred task is generated when an `if` clause is specified on a task construct and the `if` clause expression evaluates to zero.

An included task is a task whose execution is sequentially included in the generating task region. In other words, an included task is undeferred and executed immediately by the encountering thread. An included task is generated when the generated task is a child task of a final task.

A merged task is a task that has the same data environment as that of its generating task region. A merged task might be generated when both the following conditions nonzero values:

- A `mergeable` clause is specified on a task construct.
- The generated task is an undeferred task or an included task.

The `if` clause expression and the `final` clause expression are evaluated outside of the task construct, and the evaluation order cannot be specified.

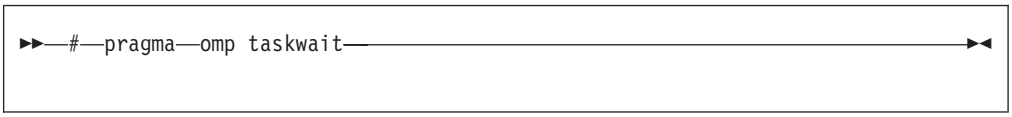
Related reference:

"#pragma omp taskwait"

#pragma omp taskwait
Purpose

Use the **taskwait** pragma to specify a *wait* for child tasks to be completed that are generated by the current task.

Syntax



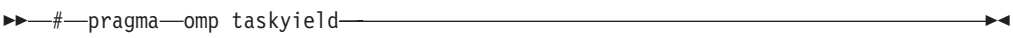
Related reference:

"#pragma omp task" on page 619

#pragma omp taskyield
Purpose

The **omp taskyield** pragma instructs the compiler to suspend the current task in favor of running a different task. The **taskyield** region includes an explicit task scheduling point in the current task region.

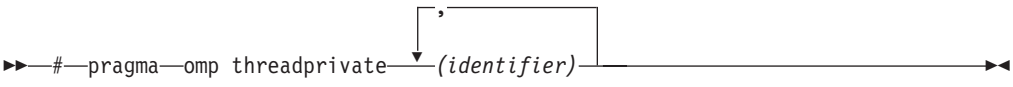
Syntax



#pragma omp threadprivate
Purpose

The **omp threadprivate** directive makes the named file-scope, namespace-scope, or static block-scope variables private to a thread.

Syntax



where *identifier* is a file-scope, name space-scope or static block-scope variable.

Usage

Each copy of an **omp threadprivate** data variable is initialized once prior to first use of that copy. If an object is changed before being used to initialize a **threadprivate** data variable, behavior is unspecified.

A thread must not reference another thread's copy of an **omp threadprivate** data variable. References will always be to the master thread's copy of the data variable when executing serial and master regions of the program.

Use of the **omp threadprivate** directive is governed by the following points:

- An **omp threadprivate** directive must appear at file scope outside of any definition or declaration.
- The **omp threadprivate** directive is applicable to static-block scope variables and may appear in lexical blocks to reference those block-scope variables. The directive must appear in the scope of the variable and not in a nested scope, and must precede all references to variables in its list.
- A data variable must be declared with file scope prior to inclusion in an **omp threadprivate** directive *list*.
- An **omp threadprivate** directive and its *list* must lexically precede any reference to a data variable found in that *list*.
- A data variable specified in an **omp threadprivate** directive in one translation unit must also be specified as such in all other translation units in which it is declared.
- Data variables specified in an **omp threadprivate** *list* must not appear in any clause other than the **copyin**, **copyprivate**, **if**, **num_threads**, and **schedule** clauses.
- The address of a data variable in an **omp threadprivate** *list* is not an address constant.
- A data variable specified in an **omp threadprivate** *list* must not have an incomplete or reference type.

Chapter 19. Compiler predefined macros

Predefined macros can be used to conditionally compile code for specific compilers, specific versions of compilers, specific environments, and specific language features.

Predefined macros fall into several categories:

- "General macros"
- "Macros related to the platform" on page 626
- "Macros related to compiler features" on page 627

"Examples of predefined macros" on page 637 show how you can use them in your code.

General macros

The following predefined macros are always predefined by the compiler. Unless noted otherwise, all the following macros are *protected*, which means that the compiler will issue a warning if you try to undefine or redefine them.

Table 46. General predefined macros

Predefined macro name	Description	Predefined value

__COUNTER__	Expands to an integer that starts from 0. The value increases by 1 each time this macro is expanded.
	
	You can use this macro with the ## operator to generate unique variable or function names. The following example shows the declaration of distinct identifiers with a single token:
	`#define CONCAT(a, b) a##b`
	`#define CONCAT_VAR(a, b) CONCAT(a, b)`
	`#define VAR CONCAT_VAR(var, __COUNTER__)`
	
	`//Equivalent to int var0 = 1;`
	`int VAR = 1;`
	
	`//Equivalent to char var1 = 'a';`
	`char VAR = 'a';`
		An integer variable that starts from 0. The value increases by 1 each time this macro is expanded.

__FUNCTION__	Indicates the name of the function currently being compiled.	A character string containing the name of the function currently being compiled.
__LIBREL__	Indicates the Language Environment library level under which the compiler is running.	The return value of a compiler call to the `librel` library function.
__ptr31__	Expands to the pointer qualifier `__ptr32`. Not protected.	__ptr32
__PTR32	Indicates that the pointer qualifier `__ptr32` is recognized. Not protected.	1

Table 46. General predefined macros (continued)

Predefined macro name	Description	Predefined value
__TARGET_LIB__	Indicates the version of the target library.	A hexadecimal string literal representing the version number of the target library. The format of the version number is hex *PVRRMMMM*, where: *P* Represents the z/OS XL C or C/C++ library product. The possible values are: • 2 for OS/390 • 4 for z/OS Release 2 and later *V* Represents the version number *RR* Represents the release number *MMMM* Represents the modification number The value of the __TARGET_LIB__ macro depends on the setting of the TARGET compiler option, which allows you to specify the runtime environment and release for the generated object module. The __TARGET_LIB__ macro is set as follows: • 0x42010000 (for zOSV2R1 TARGET suboption) • 0x410D0000 (for zOSV1R13 TARGET suboption) • 0x410C0000 (for zOSV1R12 TARGET suboption) If the TARGET suboption is specified as a hexadecimal string literal, this macro is also defined to that literal.

Table 46. General predefined macros (continued)

Predefined macro name	Description	Predefined value
__TIMESTAMP__	Indicates the date and time when the source file was last modified. The value changes as the compiler processes any include files that are part of your source program. This macro is available for partitioned data sets (PDSs/PDSEs) and z/OS UNIX files only. For PDSE or PDS members, the ISPF timestamp for the member is used if present. For PDSE/PDS members with no ISPF timestamp, sequential data sets, or in stream source in JCL, the compiler returns a dummy timestamp. For z/OS UNIX files, the compiler uses the system timestamp on a source file. Otherwise, it returns a dummy timestamp, `"Mon Jan 1 0:00:01 1990"`.	A character string literal in the form "*Day Mmm dd hh:mm:ss yyyy*", where: *Day* Represents the day of the week (`Mon`, `Tue`, `Wed`, `Thu`, `Fri`, `Sat`, or `Sun`). *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. *hh* Represents the hour. *mm* Represents the minutes. *ss* Represents the seconds. *yyyy* Represents the year.

Macros indicating the z/OS XL C/C++ compiler

Macros related to the z/OS XL C/C++ compiler are always predefined, and they are protected, which means that the compiler will issue a warning if you try to undefine or redefine them. You can specify the **SHOWMACROS(PRE)** compiler option to view predefined macro definitions in preprocessed output.

Table 47. Compiler product predefined macros

Predefined macro name	Description	Predefined value
__COMPILER_VER__	Indicates the version of the compiler.	A hexadecimal integer in the format *PVRRMMMM*, where : *P* Represents the compiler product • 0 for C/370™ • 1 for IBM AD/Cycle C/370 and C/C++ for MVS/ESA • 2 for OS/390 C/C++ • 4 for z/OS Release 2 and later *V* Represents the version number *RR* Represents the release number *MMMM* Represents the modification number

Table 47. Compiler product predefined macros (continued)

Predefined macro name	Description	Predefined value
C __IBMC__	Indicates the level of the XL C compiler.	An integer in the format *PVRRM*, where : *P* Represents the compiler product • 0 for C/370 • 1 for AD/Cycle C/370 and C/C++ for MVS/ESA • 2 for OS/390 C/C++ • 4 for z/OS Release 2 and later *V* Represents the version number *RR* Represents the release number *M* Represents the modification number C
C++ __IBMCPP__	Indicates the level of the XL C++ compiler.	An integer in the format *PVRRM*, where : *P* Represents the compiler product • 0 for C/370 • 1 for AD/Cycle C/370 and C/C++ for MVS/ESA • 2 for OS/390 C/C++ • 4 for z/OS Release 2 and later *V* Represents the version number *RR* Represents the release number *M* Represents the modification number C++

Macros related to the platform

The following predefined macros are provided to facilitate porting applications between platforms. All platform-related predefined macros are unprotected and can be undefined or redefined without warning unless otherwise specified.

Table 48. Platform-related predefined macros

Predefined macro name	Description	Predefined value	Predefined under the following conditions
__370__	Indicates that the program is compiled or targeted to run on IBM System/370.	1	Always predefined.
__HHW_370__	Indicates that the host hardware is System/370.	1	Always predefined for z/OS.
__HOS_MVS__	Indicates that the host operating system is z/OS.	1	Always predefined for z/OS.
__MVS__	Indicates that the host operating system is z/OS.	1	Always predefined for z/OS.
__THW_370__	Indicates that the target hardware is System/370.	1	Always predefined for z/OS__THW_PPC__

Table 48. Platform-related predefined macros (continued)

Predefined macro name	Description	Predefined value	Predefined under the following conditions
__TOS_MVS__	Indicates that the host operating system is z/OS.	1	Always predefined for z/OS.

Macros related to compiler features

Feature-related macros are predefined according to the setting of specific compiler options or pragmas. Unless noted otherwise, all feature-related macros are protected, which means that the compiler will issue a warning if you try to undefine or redefine them.

Feature-related macros are discussed in the following sections:

- "Macros related to compiler option settings"
- "Macros related to language levels" on page 632

Macros related to compiler option settings

The following macros can be tested for various features, including source input characteristics, output file characteristics, and optimization. All of these macros are predefined by a specific compiler option or suboption, or any invocation or pragma that implies that suboption. If the suboption enabling the feature is not in effect, then the macro is undefined.

See the description of each option in the *z/OS XL C/C++ User's Guide* for detailed information about the option.

Table 49. General option-related predefined macros

Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect:
__64BIT__	Indicates that 64-bit compilation mode is in effect.	1	-q64 or LP64
__ARCH__	Indicates the target architecture for which the source code is being compiled.	The integer value specified in the ARCH compiler option.	ARCH(*integer value*)
__BFP__	Indicates that binary floating point (BFP) mode is in effect.	1	FLOAT(IEEE)
C++ __BOOL__		0	NOKEYWORD(BOOL)
C++ __C99_RESTRICT		1	KEYWORD(RESTRICT)
_CHAR_SIGNED	Indicates that the default character type is `signed char`.	1	CHARS(SIGNED)
_CHAR_UNSIGNED	Indicates that the default character type is `unsigned char`.	1	CHARS(UNSIGNED)

Table 49. General option-related predefined macros (continued)

Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect:
__CHARSET_LIB		0	NOASCII
		1	ASCII
__CICS__	Indicates that embedded CICS statements are accepted	1	CICS
__CODESET__	Indicates the character code set in effect.	The compiler uses the following runtime function to determine the compile-time character code set: `nl_langinfo (CODESET)`	LOCALE
C++ _CPPUNWIND	Indicates that C++ exception handling is enabled.	1	EXH
__DIGRAPHS__	Indicates support for digraphs.	1	DIGRAPH
__DLL__	Indicates that the program is compiled as DLL code.	1	C DLL C++ Always predefined.
__ENUM_OPT	Indicates that the compiler supports the ENUMSIZE option and the **#pragma enum** directive.	1	Always predefined.
__FILETAG__	Indicates the character code set of the current file.	The string literal specified by the **#pragma filetag** directive. The value changes as the compiler processes include files that make up the source program.	#pragma filetag(*string literal*)
__GOFF__		1	GOFF
__IBM_ASM_SUPPORT	Indicates that inline assembly statements are supported.	1	ASM
C __IBM_FAR_IS_SUPPORTED__	Indicates that the `__far` type qualifier is supported.	1	METAL

Table 49. General option-related predefined macros (continued)

Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect:
C++ __IBM__TYPEOF__		1	KEYWORD(TYPEOF)
__IBM_DFP__	Indicates support for decimal floating-point types.	1	DFP
C __IBM_METAL__	Indicates that LE-independent HLASM code is to be generated by the compiler.	1	METAL
__IBM_UTF_LITERAL	Indicates support for UTF-16 and UTF-32 string literals.	1	LANGLVL(EXTENDED) C++ LANGLVL (EXTENDED0X)
C++ __IBMCPP_LONGNAME__	Indicates that the LONGNAME compiler option is specified (which is the default for z/OS XL C++). This macro is only defined for LONGNAME. This macro is only available starting with z/OS V1R10. Users should check their z/OS XL C++ compiler level by using __IBMCPP__ when using this macro.	1	LONGNAME
__IGNERRNO__	Indicates that system calls do not modify `errno`, thereby enabling certain compiler optimizations.	1	IGNERRNO
_ILP32	Indicates that 32-bit compilation mode is in effect.	1	-q32 or ILP32
__INITAUTO__	Indicates the value to which automatic variables which are not explicitly initialized in the source program are to be initialized.	A hexadecimal constant in the form `(0x`*nn*`U)`, including the parentheses, where *nn* represents the value specified in the INITAUTO compiler option.	INITAUTO(*hex value*)

Table 49. General option-related predefined macros (continued)

Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect:
__INITAUTO_W__	Indicates the value to which automatic variables which are not explicitly initialized in the source program are to be initialized.	A hexadecimal constant in the form `(0x`*nnnnnnnn*`U)`, including the parentheses, where *nnnnnnnn* represents one of the following: • If you specified an eight-digit ("word") value in the INITAUTO compiler option, *nnnnnnnn* is the value you specified. • If you specified a two-digit ("byte") value in the INITAUTO compiler option, *nnnnnnnn* is the two-digit value repeated 4 times.	INITAUTO(*hex value*)
_LARGE_FILES	Indicates that large file support is enabled, which allows access to hierarchical file system files that are larger than 2 gigabytes.	1	DEFINE(_LARGE_FILES)
__LIBANSI__	Indicates that calls to functions whose names match those in the C Standard Library are in fact the C library functions, enabling certain compiler optimizations.	1	LIBANSI

Table 49. General option-related predefined macros (continued)

Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect:

__LOCALE__	Contains a string literal that represents the locale of the LOCALE compiler option. The following example shows how to use the __LOCALE__ macro:
	`int main()`
	`{`
	`#ifdef __LOCALE__`
	`/* If the locale option is not specified,*/`
	`/* we can just follow the default locale*/`
	`setlocale(LC_ALL, __LOCALE__);`
	`#endif`
	`...`
	`}`
		The compiler uses the following runtime function to determine the compile-time locale:
		`setlocale (LC_ALL, "string literal");`
			LOCALE(*string literal*)

__LONGNAME__	Indicates that identifiers longer than 8 characters are allowed.	1	C LONGNAME C++ Always predefined.
_LP64	Indicates that 64-bit compilation mode is in effect.	1	-q64 or LP64
__LP64__	Indicates that 64-bit compilation mode is in effect.	1	-q64 or LP64
C++ __OBJECT_MODEL_CLASSIC__	Indicates that the "classic" object model is in effect.	1	OBJECTMODEL (CLASSIC)
C++ __OBJECT_MODEL_IBM__	Indicates that the IBM object is in effect.	1	OBJECTMODEL(IBM)
__OPTIMIZE__	Indicates the level of optimization in effect.	The integer value specified in the OPT compiler option.	OPT(*integer value*)
C++ __RTTI_DYNAMIC_CAST__	Indicates that runtime type identification (RTTI) information for the `typeid` and `dynamic_cast` operator is generated.	1	RTTI | RTTI(ALL | DYNAMICCAST)
__SQL__	Indicates that processing of embedded SQL statements is enabled.	1	SQL

Table 49. General option-related predefined macros (continued)

Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect:
C++ __TEMPINC__	Indicates that the compiler is using the template-implementation file method of resolving template functions.	1	TEMPINC
__TUNE__	Indicates the architecture for which the compiler generated executable code is optimized.	The integer value specified in the TUNE compiler option.	TUNE(*integer value*)
__VEC__	Indicates support for vector data types.	10206	VECTOR
__XPLINK__	Indicates that a z/OS linkage is in effect to increase performance.	1	XPLINK

Macros related to language levels

The following macros can be tested for C99 features, features related to GNU C or C++, and other IBM language extensions. All of these macros are predefined to a value of 1 by a specific language level, represented by a suboption of the LANGLVL compiler option, or any invocation or pragma that implies that suboption. If the suboption enabling the feature is not in effect, then the macro is undefined.

Table 50. Predefined macros for language features

Predefined macro name	Description	Predefined when the following language level is in effect
C __ANSI__	Indicates that only language constructs that support the ISO C89 Standard are allowed.	ansi | stdc89 | stdc99
C++ __BOOL__	Indicates that the `bool` keyword is accepted.	Always defined except when NOKEYWORD(bool) is in effect.
C __C99_BOOL	Indicates support for the `_Bool` data type.	extc1x | stdc99 | extc99 | extc89 | extended
C++ __C99_COMPLEX	Indicates that the support for C99 complex types is enabled or that the C99 complex header should be included.	extc1x | stdc99 | extc99 | extc89 | extended
C __C99_CPLUSCMT	Indicates support for C++ style comments	extc1x | stdc99 | extc99 | stdc89 | extc89 | extended (also SSCOM)
__C99_COMPOUND_LITERAL	Indicates support for compound literals.	extc1x | stdc99 | extc99 | extc89 | extended
C __C99_DESIGNATED_INITIALIZER	Indicates support for designated initialization.	extc1x | stdc99 | extc99 | extc89 | extended

Table 50. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
C __C99_DUP_TYPE_QUALIFIER	Indicates support for duplicated type qualifiers.	extc1x | stdc99 | extc99 | extc89 | extended
C __C99_EMPTY_MACRO_ARGUMENTS	Indicates support for empty macro arguments.	extc1x | stdc99 | extc99 | extc89 | extended
C __C99_FLEXIBLE_ARRAY_MEMBER	Indicates support for flexible array members.	extc1x | stdc99 | extc99 | extc89 | extended
__C99__FUNC__	Indicates support for the `__func__` predefined identifier.	C extc1x | stdc99 | extc99 | extc89 | extended C++ extended | extended0x |c99__func__
__C99_HEX_FLOAT_CONST	Indicates support for hexadecimal floating constants.	extc1x | stdc99 | extc99 | extc89 | extended
C __C99_INLINE	Indicates support for the `inline` function specifier.	extc1x | stdc99 | extc99
__C99_LLONG	Indicates support for C99-style `long long` data types and literals.	extc1x | stdc99 | extc99 C++ extended0x | c99longlong
__C99_MACRO_WITH_VA_ARGS	Indicates support for function-like macros with variable arguments.	extc1x | stdc99 | extc99 | extc89 | extended C++ extended
__C99_MAX_LINE_NUMBER	Indicates that the maximum line number is 2147483647.	extc1x | stdc99 | extc99 | extc89 | extended C++ extended0x | c99preprocessor
C __C99_MIXED_DECL_AND_CODE	Indicates support for mixed declaration and code.	extc1x | stdc99 | extc99 | extc89 | extended
__C99_MIXED_STRING_CONCAT	Indicates support for concatenation of wide string and non-wide string literals.	extc1x | stdc99 | extc99 | extc89 | extended C++ extended0x | c99preprocessor
C __C99_NON_LVALUE_ARRAY_SUB	Indicates support for non-lvalue subscripts for arrays.	extc1x | stdc99 | extc99 | extc89 | extended
C __C99_NON_CONST_AGGR_INITIALIZER	Indicates support for non-constant aggregate initializers.	extc1x | stdc99 | extc99 | extc89 | extended
__C99_PRAGMA_OPERATOR	Indicates support for the `_Pragma` operator.	extc1x | stdc99 | extc99 | extc89 | extended C++ extended

Table 50. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
C __C99_REQUIRE_FUNC_DECL	Indicates that implicit function declaration is not supported.	stdc99
C __C99_RESTRICT	Indicates support for the C99 `restrict` qualifier.	extc1x | stdc99 | extc99
C __C99_STATIC_ARRAY_SIZE	Indicates support for the `static` keyword in array parameters to functions.	extc1x | stdc99 | extc99 | extc89 | extended
C __C99_STD_PRAGMAS	Indicates support for standard pragmas.	extc1x | stdc99 | extc99 | extc89 | extended
C __C99_TGMATH	Indicates support for type-generic macros in tgmath.h	extc1x | stdc99 | extc99 | extc89 | extended
__C99_UCN	Indicates support for universal character names.	C extc1x | stdc99 | extc99 | extc89 | extended
C __C99_VAR_LEN_ARRAY	Indicates support for variable length arrays.	extc1x | stdc99 | extc99 | extc89 | extended
C++ __C99_VARIABLE_LENGTH_ARRAY	Indicates support for variable length arrays.	extended | extended0x | c99vla
C __COMMONC__	Indicates that language constructs defined by XPG are allowed.	commonc
C++ __COMPATMATH__	Indicates that the newer C++ function declarations are not to be introduced by the math.h header file.	oldmath
_EXT	Used in features.h to control the availability of extensions to the general ISO run-time libraries.	LIBEXT, or any LANGLVL suboption that implies it. (See the description of the LANGLVL option in the *z/OS XL C/C++ User's Guide* for a list of suboptions that imply LIBEXT.)
__EXTENDED__	Indicates that language extensions are supported.	extended
__IBM_COMPUTED_GOTO	Indicates support for computed `goto` statements.	C extc1x | extc99 | extc89 | extended C++ extended | extended0x | gnu_computedgoto
__IBM_INCLUDE_NEXT	Indicates support for the `#include_next` preprocessing directive.	Always defined.
__IBM_LABEL_VALUE	Indicates support for labels as values.	C extc1x | extc99 | extc89 | extended C++ extended | extended0x |gnu_labelvalue

Table 50. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
__IBM__TYPEOF__	Indicates support for the `__typeof__ or typeof` keyword.	C Always defined C++ extended | extended0x (Also KEYWORD(TYPEOF))
__IBMC_COMPLEX_INIT	Indicates support for the initialization of complex types: float _Complex, double _Complex, and long double _Complex.	extc1x
__IBMC_GENERIC	Indicate support for the generic selection feature.	C extc89 | extc99 | extended | extc1x
__IBMC_NORETURN	Indicates support for the `_Noreturn` function specifier.	C extc89 | extc99 | extended | extc1x C++ extended | extended0x | c1xnoreturn
C11 __IBMC_STATIC_ASSERT	Indicates support for the static assertions feature.	extc89 | extc99 | extended | extc1x
C++11 __IBMCPP_AUTO_TYPEDEDUCTION	Indicates support for the auto type deduction feature.	extended0x | autotypededuction
C++11 __IBMCPP_C99_LONG_LONG	Indicates support for the C99 `long long` feature.	extended0x | c99longlong
C++11 __IBMCPP_C99_PREPROCESSOR	Indicates support for the C99 preprocessor features adopted in the C++11 standard.	extended0x | c99preprocessor
__IBMCPP_COMPLEX_INIT	Indicates support for the initialization of complex types: float _Complex, double _Complex, and long double _Complex.	extended
C++11 __IBMCPP_CONSTEXPR	Indicates support for the generalized constant expressions feature.	extended0x | constexpr
C++11 __IBMCPP_DECLTYPE	Indicates support for the decltype feature.	extended0x | decltype
C++11 __IBMCPP_DEFAULTED_AND_DELETED_FUNCTIONS	Indicates support for the defaulted and deleted functions feature.	extended0x | defaultanddelete
C++11 __IBMCPP_DELEGATING_CTORS	Indicates support for the delegating constructors feature.	extended0x | delegatingctors
C++11 __IBMCPP_EXPLICIT_CONVERSION_OPERATORS	Indicates support for the explicit conversion operators feature.	extended0x | explicitconversionoperators

Table 50. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
C++11 __IBMCPP_EXTENDED_FRIEND	Indicates support for the extended friend declarations feature.	extended0x | extendedfriend
C++11 __IBMCPP_EXTERN_TEMPLATE	Indicates support for the explicit instantiation declarations feature.	extended | extended0x | externtemplate
C++11 __IBMCPP_INLINE_NAMESPACE	Indicates support for the inline namespace definitions feature.	extended0x | inlinenamespace
C++11 __IBMCPP_REFERENCE_COLLAPSING	Indicates support for the reference collapsing feature.	extended0x | referencecollapsing
C++11 __IBMCPP_RIGHT_ANGLE_BRACKET	Indicates support for the right angle bracket feature.	extended0x | rightanglebracket
C++11 __IBMCPP_RVALUE_REFERENCES	Indicates support for the rvalue references feature.	extended0x | rvaluereferences
C++11 __IBMCPP_SCOPED_ENUM	Indicates support for the scoped enumeration feature.	extended0x | scopedenum
C++11 __IBMCPP_STATIC_ASSERT	Indicates support for the static assertions feature.	C++ extended0x | static_assert
C++11 __IBMCPP_VARIADIC_TEMPLATES	Indicates support for the variadic templates feature.	extended0x | variadic[templates]
_LONG_LONG	Indicates support for `long long` data types.	C extc1x | stdc99 | extc99 | | stdc89 | extc89 | extended | longlong C++ extended0x | c99longlong | extended | longlong
_MI_BUILTIN	Indicates that the machine instruction built-in functions are available.	LIBEXT, or any LANGLVL suboption that implies it. (See the description of the LANGLVL option in the *z/OS XL C/C++ User's Guide* for a list of suboptions that imply LIBEXT.)
C __RESTRICT__	Indicates that the `__restrict__` or `__restrict` keywords are supported.	Predefined at all language levels.
C __SAA__	Indicates that only language constructs that support the most recent level of SAA C standards are allowed.	saa

Table 50. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
C __SAA_L2__	Indicates that only language constructs that conform to SAA Level 2 C standards are allowed.	saal2

Examples of predefined macros

This example illustrates use of the __FUNCTION__ and the __C99__FUNC__ macros to test for the availability of the C99 __func__ identifier to return the current function name:

```
#include <stdio.h>

#if defined(__C99__FUNC__)
#define PRINT_FUNC_NAME() printf (" In function %s \n", __func__);
#elif defined(__FUNCTION__)
#define PRINT_FUNC_NAME() printf (" In function %s \n", __FUNCTION__);
#else
#define PRINT_FUNC_NAME() printf (" Function name unavailable\n");
#endif

void foo(void);

int main(int argc, char **argv)
{
   int k = 1;
   PRINT_FUNC_NAME();
   foo();
   return 0;
}

void foo (void)
{
   PRINT_FUNC_NAME();
   return;
}
```

The output of this example is:

```
In function main
In function foo
```

C++ This example illustrates use of the __FUNCTION__ macro in a C++ program with virtual functions.

CCNX08C

```
#include <stdio.h>
class X { public: virtual void func() = 0;};

class Y : public X {
   public: void func() { printf("In function %s \n", __FUNCTION__);}
};

int main() {
  Y aaa;
  aaa.func();
}
```

The output of this example is:

```
In function Y::func()
```

Chapter 20. The IBM XL C/C++ language extensions

The IBM XL C++ extensions include C++ features as extensions in the following categories:

General IBM extensions

The following feature is enabled by default at all extended language levels:

Table 51. General IBM extensions

Language feature	Discussed in:
Allowed types in `typedef` definitions	"typedef definitions" on page 74

The following feature is disabled by default at all language levels. It also can be enabled or disabled by an individual option.

Table 52. General IBM extensions with individual option controls

Language feature	Discussed in:	Individual option controls
A non-const or volatile lvalue reference bound to an rvalue of a user-defined type	"Initialization of references (C++ only)" on page 119	LANGLVL([NO] COMPATRVALUEBINDING)
Extra text after #endif or #else	"Extension of #endif and #else (IBM extension)" on page 522	LANGLVL([NO]TEXTAFTERENDIF)

The following feature is enabled with the LANGLVL(COMPAT366 | EXTENDED) option. It can also be enabled or disabled by a specific compiler option, listed in the following table:

Table 53. General IBM extensions with individual option controls

Language feature	Discussed in:	Individual option controls
Non-C99 IBM `long long` extension	"Types of integer literals outside of C99 and C++11" on page 20	[NO]LONGLONG

The following feature is enabled with the LANGLVL(EXTENDED0X) option. It can also be enabled or disabled by specific compiler options, listed in the following table:

Table 54. General IBM extensions with individual option controls

Language feature	Discussed in:	Individual option controls
C99 `long long` feature with the associated IBM extensions	Types of integer literals in C99 and C++11	C++11 LANGLVL([NO]C99LONGLONG), IBM LANGLVL([NO] EXTENDEDINTEGERSAFE)

Extensions for C11 compatibility

Note: C11 is a new version of the C programming language standard. IBM continues to develop and implement the features of the new standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the features of the C11 standard is complete, including the support of a new C standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility with earlier releases, in source, binary, or options, of IBM's implementation of the new features of the C11 standard and therefore they should not be relied on as a stable programming interface.

The following features are part of a continual phased release process leading towards full compliance with C11. They can be enabled by using the LANGLVL(EXTC1X) group option.

Table 55. IBM XL C language extensions for compatibility with C11

Language feature	Discussed in:
Anonymous structures	"Anonymous structures (C11)" on page 68
Complex type initialization	"Initialization of complex types (C11)" on page 123
Generic selection	"Generic selection (C11)" on page 146
Static assertions	"_Static_assert declaration (C11)" on page 45
The `_Noreturn` function specifier	"The _Noreturn function specifier" on page 230

C++11 compatibility

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

The following features are part of a continual phased release process leading towards full compliance with C++11. They can be enabled by using the

LANGLVL(EXTENDED0X) group option. You can also use specific compiler options to enable or disable these features. See the following table.

Table 56. Supported C++11 features

Language feature	Discussed in:	C++11 individual suboption control
Auto type deduction	"The auto type specifier (C++11)" on page 76	**LANGLVL([NO]AUTOTYPEDEDUCTION)**
C99 `long long`	"Types of integer literals in C99 and C++11" on page 21	**LANGLVL([NO]C99LONGLONG)**, IBM **LANGLVL([NO]EXTENDEDINTEGERSAFE)**
C99 preprocessor features adopted in C++11	"C99 preprocessor features adopted in C++11" on page 526	**LANGLVL([NO]C99PREPROCESSOR)**
Decltype	"The decltype(*expression*) type specifier (C++11)" on page 78	**LANGLVL([NO]DECLTYPE)**
Defaulted and deleted functions	"Explicitly defaulted functions" on page 221 "Deleted functions" on page 222	**LANGLVL([NO]DEFAULTANDDELETE)**
Delegating constructors	"Delegating constructors (C++11)" on page 409	**LANGLVL([NO]DELEGATINGCTORS)**
Explicit conversion operators	"Explicit conversion operators (C++11)" on page 427	**LANGLVL([NO]EXPLICITCONVERSIONOPERATORS)**
Explicit instantiation declarations	" Explicit instantiation declaration" on page 456	**LANGLVL([NO]EXTERNTEMPLATE)**
Extended friend declarations	"Friends" on page 373	**LANGLVL([NO]EXTENDEDFRIEND)**
Forward declaration of enumerations	"Enumerations" on page 70	N/A
Generalized constant expressions	"Generalized constant expressions (C++11)" on page 149	**LANGLVL([NO]CONSTEXPR)**
Inline namespace definitions	"Inline namespace definitions (C++11)" on page 324	**LANGLVL([NO]INLINAMESPACE)**
Reference collapsing	"Reference collapsing (C++11)" on page 194	**LANGLVL([NO]REFERENCECOLLAPSING)**
Right angle brackets	"Class templates" on page 441	**LANGLVL([NO]RIGHTANGLEBRACKET)**
Rvalue references	Using rvalue reference (C++11) in *z/OS XL C/C++ Programming Guide*	**LANGLVL([NO]RVALUEREFERENCES)**

Table 56. Supported C++11 features (continued)

Language feature	Discussed in:	C++11 individual suboption control
Scoped enumerations	"Enumerations" on page 70	**LANGLVL([NO]SCOPEDENUM)**
`static_assert`	"static_assert declaration (C++11)" on page 46	**LANGLVL([NO]STATIC_ASSERT)**
Trailing comma allowed in `enum` declarations	"Enumerations" on page 70	**LANGLVL([NO]TRAILENUM)**
Trailing return type	"Trailing return type (C++11)" on page 239	**LANGLVL([NO]AUTOTYPEDEDUCTION)**
Variadic templates	"Variadic templates (C++11)" on page 468	**LANGLVL([NO]VARIADIC[TEMPLATES])**

Notes:

- You can also use the **LANGLVL(EXTENDED)** group option to enable the explicit instantiation declarations feature.
- If you try to use a C++11 feature when the feature is not enabled, the compiler issues an information message that follows a syntax error message. The information message indicates how to turn on the C++11 feature to recover from the syntax error. The involved C++11 features are listed as follows:
  - C99 preprocessor features adopted in C++11
    - Mixed string literal concatenation
    - The `__STDC_HOSTED__` macro
  - Defaulted and deleted functions
  - Delegating constructors
  - Explicit conversion operators
  - Generalized constant expressions
  - Inline namespace definitions
  - `nullptr`
  - Reference collapsing
  - Right angle brackets
  - Rvalue references
  - Scoped enumerations
  - Variadic templates

Extensions for GNU C/C++ compatibility

The following subset of the GNU C/C++ language extension is enabled with the **LANGLVL(EXTENDED)** option, which is the default language level.

Table 57. Default IBM XL C/C++ extensions for compatibility with GNU C/C++

Language feature	Discussed in:
Placement of flexible array members anywhere in structure or union	Flexible array members (C only)
Static initialization of flexible array members of aggregates	Flexible array members (C only)

Table 57. Default IBM XL C/C++ extensions for compatibility with GNU C/C++ (continued)

Language feature	Discussed in:
`__alignof__` operator	"The __alignof__ operator (IBM extension)" on page 156
`__typeof__` operator	"The typeof operator (IBM extension)" on page 159
Generalized lvalues	"Lvalues and rvalues" on page 141
Complex type arguments to unary operators	"Unary expressions" on page 151
`__imag__` and `__real__` complex type operators	"The __real__ and __imag__ operators (IBM extension)" on page 160
Function attributes	"Function attributes (IBM extension)" on page 242

Some extensions can also be enabled or disabled by specific compiler options, which are listed in the following table:

Table 58. Default IBM XL C/C++ extensions for compatibility with GNU C/C++, with individual option controls

Language feature	Discussed in:	Individual option controls
Labels as values	"Labels as values (IBM extension)" on page 198	**LANGLVL([NO]GNU_LABELVALUE)**
Computed `goto` statements	"Computed goto statement (IBM extension)" on page 214	**LANGLVL([NO]GNU_ COMPUTEDGOTO)**

Extensions for Unicode support

The ISO C and ISO C++ Committees have approved the implementation of *u-literals* and *U-literals* to support Unicode UTF-16 and UTF-32 character literals, respectively. They are enabled under the **EXTENDED** and **EXTENDED0X** language levels.

Table 59. IBM XL C++ extensions for unicode support

Language feature	Discussed in:	Required compilation option
UTF-16, UTF-32 literals	"UTF literals (IBM extension)" on page 36	**LANGLVL(EXTENDED0X)**

Extensions for vector processing support

XL C/C++ V2R1M1 web deliverable for z/OS 2.1 provides vector programming support for programmers to make use of the Vector Facility for z/Architecture, which is based on the AltiVec Programming Interface specification with suitable changes and extensions.

See Using vector programming support in *z/OS XL C/C++ Programming Guide, SC14-7315* for detailed information about the language extensions for vector processing support, including compiler options, vector data types and operators, macro, and built-in functions.

Appendix A. C and C++ compatibility on the z/OS platform

This information pertains to the differences between C and C++ that apply specifically to the z/OS platform. The contents describe the constructs that are found in both ISO C and ISO C++, but which are treated differently in the two languages.

String initialization

In C++, when you initialize character arrays, a trailing `'\0'` (zero of type `char`) is appended to the string initializer. You cannot initialize a character array with more initializers than there are array elements.

In C, space for the trailing `'\0'` can be omitted in this type of initialization.

The following initialization, for instance, is not valid in C++:

```
char v[3] = "asd"; /* not valid in C++, valid in C */
```

because four elements are required. This initialization produces an error because there is no space for the implied trailing `'\0'` (zero of type `char`).

Class/structure and typedef names

In C++, a class or structure and a `typedef` cannot both use the same name to refer to a different type within the same scope (unless the `typedef` is a synonym for the class or structure name). In C, a `typedef` name and a `struct` tag name declared in the same scope can have the same name because they have different name spaces. For example:

```
int main ()
{
     typedef double db;
     struct db; /* error in C++, valid in C */

     typedef struct st st; /* valid C and C++ */
}
```

The same distinction applies within class/structure declarations. For example:

```
int main ()
{
     typedef double db;
     struct st
     {
          db x;
          double db; /* error in C++, valid in C */
     };
}
```

Class/structure and scope declarations

In C++, a class declaration introduces the class or structure name into the scope where it is declared and hides any object, function, or other declaration of that name in an outer scope. In C, an inner scope declaration of a `struct` name does not hide an object or function of that name in an outer scope. For example:

```
double db;
int main ()
{
     struct db               /* hides double object db in C++ */
     { char* str; };
     int x = sizeof(db);     /* size of struct in C++ */
                             /* size of double in C */
}
```

const object initialization

In C++, `const` objects must be initialized. In C, they can be left uninitialized.

Definitions

An object declaration is a definition in C++. In C, it is a declaration (also known as a tentative definition). For example:

```
int i;
```

In C++, a global data object must be defined only once. In C, a global data object can be declared several times without using the `extern` keyword.

In C++, multiple definitions for a single variable cause an error. A C compilation unit can contain many identical declarations for a variable.

Definitions within return or argument types

In C++, types may not be defined in return or argument types. C allows such definitions. For example, the following declarations produce errors in C++, but are valid declarations in C:

```
void print(struct X { int i;} x);      /* error in C++ */
enum count{one, two, three} counter(); /* error in C++ */
```

Enumerator type

An enumerator has the same type as its enumeration in C++. In C, an enumeration has type `int`.

Enumeration type

The assignment to an object of enumeration type with a value that is not of that enumeration type produces an error in C++. In C, an object of enumeration type can be assigned values of any integral type.

Function declarations

In C++, all declarations of a function must match the unique definition of a function. C has no such restriction.

Functions with an empty argument list

Consider the following function declaration:

```
int f();
```

In C++, this function declaration means that the function takes no arguments. In C, it could take any number of arguments, of any type.

Global constant linkage

In C++, an object declared `const` has internal linkage, unless it has previously been given external linkage. In C, it has external linkage.

Jump statements

C++ does not allow you to jump over declarations containing initializations. C does allow you to use jump statements for this purpose.

Keywords

C++ contains some additional keywords not found in C. C programs that use these keywords as identifiers are not valid C++ programs:

Table 60. C++ keywords

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

main() recursion

In C++, `main()` cannot be called recursively and cannot have its address taken. C allows recursive calls and allows pointers to hold the address of `main()`.

Names of nested classes/structures

In C++, the name of a nested class is local to its enclosing class. In C, the name of the nested structure belongs to the same scope as the name of the outermost enclosing structure.

Pointers to void

C++ allows `void` pointers to be assigned only to other `void` pointers. In C, a pointer to `void` can be assigned to a pointer of any other type without an explicit cast.

Prototype declarations

C++ requires full prototype declarations. C allows nonprototyped functions.

Return without declared value

In both C and C++, the function `main()` must be declared to return a value of type `int`. In C++, if no value is explicitly returned from function `main()` by means of a return statement and if program execution reaches the end of function `main()` (that is, the program does not terminate due to a call to`exit()`, `std::terminate()`, or a similar function), then the value 0 is implicitly returned. A return (either explicit or implicit) from all other functions that are declared to return a value *must* return a value. In C, a function that is declared to return a value can return with no value, with unspecified results.

__STDC__ macro

The predefined macro variable __STDC__ is defined for C++, and it has the integer value 0 when it is used in an #if statement, indicating that the C++ language is not a proper superset of C, and that the compiler does not conform to C. In C, __STDC__ has the integer value 1.

Appendix B. Common Usage C language level for the z/OS platform

The X/Open Portability Guide (XPG) Issue 3 describes a C language definition referred to as Common Usage C. This language definition is roughly equivalent to K&R C, and differs from the ISO C language definition. It is based on various C implementations that predate the ISO standard.

Common Usage C is supported with the LANGLVL(COMMONC) compiler option or the `#pragma langlvl(commonc)` directive. These cause the compiler to accept C source code containing Common Usage C constructs.

Many of the Common Usage C constructs are already supported by `#pragma langlvl(extended)`. The following language elements are different from those accepted by `pragma langlvl(extended)`.

- Standard integral promotions preserve sign. For example, `unsigned char` or `unsigned short` are promoted to `unsigned int`. This is functionally equivalent to specifying the UPCONV compiler option.
- Trigraphs are not processed in string or character literals. For example, consider the following source line:

  ```
  ??=define STR "??= not processed"
  ```

  The above line gets preprocessed to:

  ```
  #define STR "??= not processed"
  ```
- The `sizeof` operator is permitted on bitfields. The result is the size of an `unsigned int` (4).
- Bitfields other than type `int` are permitted. The compiler issues a warning and changes the type to `unsigned int`.
- Macro parameters found within single or double quotation marks are expanded. For example, consider the following source lines:

  ```
  #define STR(AAA) "String is: AAA"
  #define ST STR(BBB)
  ```

  The above lines are preprocessed to:

  ```
  "String is: BBB"
  ```
- Macros can be redefined without first being undefined (that is, without an intervening `#undef`). An informational message is issued saying that the second definition is used.
- The empty comment (`/**/`) in a function-like macro is equivalent to the ISO token concatenation operator `##`.

The LANGLVL compiler option is described in *z/OS XL C/C++ User's Guide*. The `#pragma langlvl` is described in "#pragma langlvl (C only)" on page 564.

Appendix C. Conforming to POSIX 1003.1

The implementation resulting from the combination of z/OS UNIX System Services and the z/OS Language Environment supports the ISO/IEC 9945-1:1990/IEEE POSIX 1003.1-1990 standard. POSIX stands for Portable Operating System Interface.

For a description of how the z/OS UNIX System Services implementation meets the criteria, see IBM Corporation Conformance Statement.

Appendix D. Implementation-defined behavior

This information describes how the z/OS XL C/C++ compilers define some of the implementation-specific behavior from the ISO C and C++ standards. In-depth usage information is provided in *z/OS XL C/C++ User's Guide* and *z/OS XL C/C++ Programming Guide*.

Identifiers

The number of significant characters in an identifier with no external linkage:

- 1024

The number of significant characters in an identifier with external linkage:

- 1024 with the compile-time option LONGNAME specified
- 8 otherwise

  The C++ compiler truncates external identifiers without C++ linkage after 8 characters if the NOLONGNAME compiler option or pragma is in effect.

Case sensitivity of external identifiers:

- The binder accepts all external names up to 1024 characters, and is optionally case sensitive. The linkage editor accepts all external names up to 8 characters, and may not be case sensitive, depending on whether you use the NOLONGNAME compiler option or pragma. When the NOLONGNAME option is used, all external names are truncated to 8 characters. As an aid to portability, identifiers that are not unique after truncation are flagged as an error.

Characters

Source and execution characters which are not specified by the ISO standard:

- The caret (^) character in ASCII (bitwise exclusive OR symbol) or the equivalent not (¬) character in EBCDIC.
- The vertical broken line (¦) character in ASCII which may be represented by the vertical line (|) character on EBCDIC systems.

Shift states used for the encoding of multibyte characters:

- The shift states are indicated with the SHIFTOUT (hex value \x0E) characters and SHIFTIN (hex value \x0F).

The number of bits that represent a single-byte character:

- 8 bits

The mapping of members of the source character set (characters and strings) to the execution character set:

- The same code page is used for the source and execution character set.

The value of an integer character constant that contains a character/escape sequence not represented in the basic execution character set:

- A warning is issued for an unknown character/escape sequence and the `char` is assigned the character following the back slash.

The value of a wide character constant that contains a character/escape sequence not represented in the extended execution character set:

- A warning is issued for the unknown character/escape sequence and the `wchar_t` is assigned the wide character following the back slash.

The value of an integer character constant that contains more than one character:

- The lowest four bytes represent the character constant.

The value of a wide character constant that contains more than one multibyte character:

- The lowest four bytes of the multibyte characters are converted to represent the wide character constant.

Equivalent type of `char`: `signed char`, `unsigned char`, or user-defined:

- The default for `char` is `unsigned`

Sequence of white-space characters (excluding the new-line):

- Any spaces or comments in the source program are interpreted as one space.

String conversion

Additional implementation-defined sequence forms that can be accepted by `strtod`, `strtol` and `strtoul` functions in other than the C locale:

- None

Integers

Type	Amount of storage	Range (in `limits.h`)
`signed short`	2 bytes	-32,768 to 32,767
`unsigned short`	2 bytes	0 to 65,535
`signed int`	4 bytes	-2,147,483,647 minus 1 to 2,147,483,647
`unsigned int`	4 bytes	0 to 4,294,967,295
`signed long`	4 bytes	-2,147,483,647 minus 1 to 2,147,483,647
`unsigned long`	4 bytes	0 to 4,294,967,295
`signed long long`	8 bytes	-9,223,372,036,854,775,807 minus 1 to 9,223,372,036,854,775,807
`unsigned long long`	8 bytes	0 to 18,446,744,073,709,551,615

The result of converting an integer to a `signed char`:

- The lowest 1 byte of the integer is used to represent the `char`

The result of converting an integer from a shorter signed integer:

- The lowest 2 bytes of the integer are used to represent the `short int`.

The result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented:

- The bit pattern is preserved and the sign bit has no significance.

The result of bitwise operations (`|`, &, ^) on `signed int`:

- The representation is treated as a bit pattern and 2's complement arithmetic is performed.

The sign of the remainder of integer division if either operand is negative:

- The remainder is negative if exactly one operand is negative.

The result of a right shift of a negative-valued signed integral type:

- The result is sign-extended and the sign is propagated.

Floating-point numbers

Type	Amount of storage	Range (approximate)	
		IBM z/Architecture hexadecimal format	IEEE binary format
`float`	4 bytes	5.5×10^{-79} to 7.2×10^{75}	1.2×10^{-38} to 3.4×10^{38}
`double`	8 bytes	5.5×10^{-79} to 7.2×10^{75}	2.2×10^{-308} to 1.8×10^{308}
`long double`	16 bytes	5.5×10^{-79} to 7.2×10^{75}	3.4×10^{-4932} to 1.2×10^{4932}

The following is the direction of truncation (or rounding) when you convert an integer number to an IBM z/Architecture hexadecimal floating-point number, or to an IEEE binary floating-point number:

- IBM z/Architecture hexadecimal format:

  When the floating-point cannot exactly represent the original value, the value is truncated.

  When a floating-point number is converted to a narrower floating-point number, the floating-point number is truncated.
- IEEE binary format:

  The rounding direction is determined by the ROUND compiler option. The ROUND option only affects the rounding of floating-point values that the z/OS XL C/C++ compiler can evaluate at compile time. It has no effect on rounding at run time.

C/C++ data mapping

The z/Architecture has the following boundaries in its memory mapping:

- Byte
- Halfword
- Fullword
- Doubleword

The code that is produced by the C/C++ compiler places data types on natural boundaries. Some examples are:

- Byte boundary for `char`, `_Bool/bool`, and C `decimal (n,p)` C
- Halfword boundary for `short int`
- Fullword boundary for `int`, `long int`, pointers, `float`, and C `float _Complex` C
- Doubleword boundary for `double`, `long double`, C `double _Complex`, and `long double _Complex` C

For each external defined variable, the z/OS XL C/C++ compiler defines a writeable static data instance of the same name. The compiler places other external variables, such as those in programs that you compile with the NORENT compiler option, in separate csects that are based on their names.

Arrays and pointers

The type of `size_t`:

- `unsigned int` in 32–bit mode
- `unsigned long` in 64–bit mode

The type of `ptrdiff_t`:

- `int` in 32–bit mode
- `long` in 64–bit mode

The result of casting a pointer to an integer:

- The bit patterns are preserved.

The result of casting an integer to a pointer:

- The bit patterns are preserved.

Registers

The effect of the `register` storage class specifier on the storage of objects in registers:

- The register storage class indicates to the compiler that a variable in a block scope data definition or a parameter declaration is heavily used (such as a loop control variable). It is equivalent to `auto`, except that the compiler might, if possible, place the variable into a machine register for faster access.

Structures, unions, enumerations, bit fields

The result when a member of a union object is accessed using a member of a different type:

- The result is undefined.

The alignment/padding of structure members:

- If the structure is not packed, then padding is added to align the structure members on their natural boundaries. If the structure is packed, no padding is added.

The padding at the end of structure/union:

- Padding is added to end the structure on its natural boundary. The alignment of the structure or union is that of its strictest member.

The type of an `int` bit field (`signed int`, `unsigned int`, user defined):

- The default is `unsigned`.

The order of allocation of bit fields within an `int` :

- Bit fields are allocated from low memory to high memory. For example, `0x12345678` would be stored with byte 0 containing `0x12`, and byte 3 containing `0x78`.

The rule for bit fields crossing a storage unit boundary:

- Bit fields can cross storage unit boundaries.

The integral type that represents the values of an enumeration type:

- Enumerations can have the type `char`, `short` or `long` and be either `signed` or `unsigned` depending on their smallest and largest values.

Declarators

The maximum number of declarators (pointer, array, function) that can modify an arithmetic, structure, or union type:

- The only constraint is the availability of system resources.

Statements

The maximum number of `case` values in a `switch` statement:

- Because the `case` values must be integers and cannot be duplicated, the limit is `INT_MAX`.

Preprocessing directives

Value of a single-character constant in a constant expression that controls conditional inclusion:

- Matches the value of the character constant in the execution character set.

Such a constant may have a negative value:

- Yes

The method of searching include source files (`<...>`):

- See *z/OS XL C/C++ User's Guide*.

The method of searching quoted source files:

- User include files can be specified in double quotes. See *z/OS XL C/C++ User's Guide*.

The mapping between the name specified in the include directive and the external source file name:

- See *z/OS XL C/C++ User's Guide*.

The definitions of `__DATE__` and `__TIME__` when date and time of translation is not available:

- For z/OS XL C/C++, the date and time of translation are always available.

Translation limits

System-determined means that the limit is determined by your system resources.

Table 61. Translation Limits

Nesting levels of:

Table 61. Translation Limits (continued)

• Compound statements	• System-determined
• Iteration control	• System-determined
• Selection control	• System-determined
• Conditional inclusion	• System-determined
• Parenthesized declarators	• System-determined
• Parenthesized expression	• System-determined
Number of pointer, array and function declarators modifying an arithmetic a structure, a union, and incomplete type declaration	• System-determined
Significant initial characters in:	
• Internal identifiers	• 1024
• Macro names	• 1024
• C external identifiers (*without* LONGNAME)	• 8
• C external identifiers (*with* LONGNAME)	• 1024
• C++ external identifiers	• 1024
Number of:	
• External identifiers in a translation unit	• System-determined
• Identifiers with block scope in one block	• System-determined
• Macro identifiers simultaneously declared in a translation unit	• System-determined
• Parameters in one function definition	• System-determined
• Arguments in a function call	• System-determined
• Parameters in a macro definition	• System-determined
• Parameters in a macro invocation	• 32760 under MVS
• Characters in a logical source line	• 32K minus 1
• Characters in a string literal	• `LONG_MAX` (See **1**)
• Bytes in an object	• `SHRT_MAX`
• Nested include files	• System-determined
• Enumeration constants in an enumeration	• System-determined
• Levels in nested structure or union	

Note:

1. `LONG_MAX` is the limit for automatic variables only. For all other variables, the limit is 16 megabytes.

Appendix E. Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size

Using assistive technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/OS. Consult the assistive technology documentation for specific information when using such products to access z/OS interfaces.

Keyboard navigation of the user interface

Users can access z/OS user interfaces using TSO/E or ISPF. Refer to *z/OS TSO/E Primer*, *z/OS TSO/E User's Guide*, and *z/OS ISPF User's Guide Vol I* for information about accessing TSO/E and ISPF interfaces. These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

z/OS information

z/OS information is accessible using screen readers with the BookServer/Library Server versions of z/OS books in the Internet library at:

`http://www.ibm.com/systems/z/os/zos/bkserv/`

Notices

This information was developed for products and services offered in the U.S.A. or elsewhere.

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 character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, 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:

Site Counsel
IBM Corporation
2455 South Road
Poughkeepsie, NY 12601-5400
USA

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.

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

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

COPYRIGHT LICENSE:

This information might contain sample application programs in source language, which illustrate 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. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Policy for unsupported hardware

Various z/OS elements, such as DFSMS, HCD, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted

for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see:
- For information about currently-supported IBM hardware, contact your IBM representative.

Programming interface information

This publication documents *intended* Programming Interfaces that allow the customer to write z/OS XL C/C++ programs.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Adobe, Acrobat, PostScript and all Adobe-based trademarks are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

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

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

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

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

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

Standards

The following standards are supported in combination with the Language Environment element:

- The C language is consistent with *Programming languages - C (ISO/IEC 9899:1999)* and a subset of *Programming languages - C (ISO/IEC 9899:2011)*. For more information on ISO, visit their website at http://www.iso.org.

- The C++ language is consistent with *Programming languages - C++ (ISO/IEC 14882:1998)*, *Programming languages - C++ (ISO/IEC 14882:2003(E))*, and a subset of *Programming languages - C++ (ISO/IEC 14882:2011)*.

The following standards are supported in combination with the Language Environment and z/OS UNIX System Services elements:

- A subset of *IEEE Std. 1003.1-2001 (Single UNIX Specification, Version 3).* For more information on IEEE, visit their website at http://www.iso.org.
- *IEEE Std 1003.1—1990, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C language]*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.
- The core features of *IEEE P1003.1a Draft 6 July 1991, Draft Revision to Information Technology—Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*, copyright 1992 by the Institute of Electrical and Electronic Engineers, Inc.
- *IEEE Std 1003.2—1992, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 2: Shells and Utilities*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.
- The core features of *IEEE Std P1003.4a/D6—1992, IEEE Draft Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment 2: Threads Extension [C language]*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.
- The core features of *IEEE 754-1985 (R1990) IEEE Standard for Binary Floating-Point Arithmetic (ANSI)*, copyright 1985 by the Institute of Electrical and Electronic Engineers, Inc.
- *X/Open CAE Specification, System Interfaces and Headers, Issue 4 Version 2*, copyright 1994 by The Open Group
- *X/Open CAE Specification, Networking Services, Issue 4*, copyright 1994 by The Open Group
- *X/Open Specification Programming Languages, Issue 3, Common Usage C*, copyright 1988, 1989, and 1992 by The Open Group
- United States Government's *Federal Information Processing Standard (FIPS) publication for the programming language C, FIPS-160*, issued by National Institute of Standards and Technology, 1991

Index

Special characters

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Product Number: 5650-ZOS

Printed in USA

SC14-7308-01