IBM XL C for AIX, V10.1

Language Reference

Version 10.1

IBM XL C for AIX, V10.1

Language Reference

Version 10.1

First edition

This edition applies to IBM XL C for AIX, V10.1 (Program number 5724-U80) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Contents

About this information

This information describes the syntax, semantics, and IBM® XL C Enterprise Edition for AIX® implementation of the C programming language. Although the XL C compiler conforms to the specifications maintained by the ISO standards for the C programming language, it also incorporates many extensions to the core language. These extensions have been implemented with the aims of enhancing compatibility with other compilers, and supporting new hardware capabilities. For example, many language constructs have been added for compatibility with the GNU C compiler, to maximize portability between the two development environments.

Who should read this information

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

How to use this information

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

- Standard C library functions and headers. For information on the standard C library, refer to your AIX operating system information.
- Constructs for writing multi-threaded programs, including IBM SMP directives, OpenMP directives and functions and POSIX Pthread functions. For reference information on IBM SMP and OpenMP constructs, see the *XL C Compiler Reference*; for information on Pthreads library functions, refer to your AIX information.
- Compiler pragmas, predefined macros, and built-in functions. These are described in the *XL C Compiler 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 1 through 8 discuss language elements , including lexical elements, data types, declarations, declarators, type conversions, expressions, operators, statements, and functions. Throughout these chapters, both standard features and extensions are discussed. Chapter 9 discusses directives to the preprocessor.

The last chapters in this information provide summary lists of all the extended features supported.

Conventions

Typographical conventions

The following table explains the typographical conventions used in the IBM XL C for AIX, V10.1 information.

Table 1. Typographical conventions

Typeface	Indicates	Example
bold	Lowercase commands, executable names, compiler options, and directives.	The compiler provides basic invocation commands, **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.
<u>underlining</u>	The default setting of a parameter of a compiler option or directive.	nomaf | <u>maf</u>
`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` `-03`.

Syntax diagrams

Throughout this information, diagrams illustrate XL 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):

  ►►──keyword──*required_argument*──────────────────────────────────►◄

- Optional items are shown below the main path:

  ►►──keyword──┬──────────────────────┬──────────────────────────────►◄
             └─*optional_argument*─┘

- 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.

```
►►──keyword──┬─required_argument1─┬──────────────────────────────────►◄
             └─required_argument2─┘
```
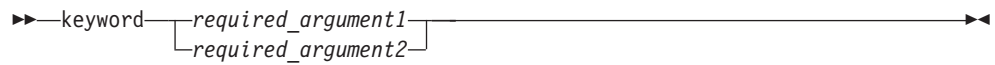
If choosing one of the items is optional, the entire stack is shown below the main path.

```
►►──keyword──┬────────────────────┬──────────────────────────────────►◄
             ├─optional_argument1─┤
             └─optional_argument2─┘
```
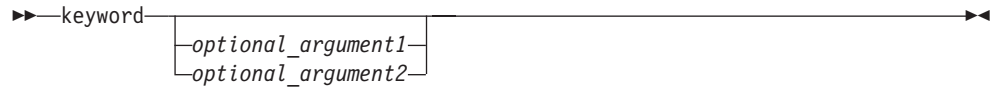
- An arrow returning to the left above the main line (a repeat arrow) indicates that you can make more than one choice from the stacked items or repeat an item. The separator character, if it is other than a blank, is also indicated:

```
          ┌──────,──────┐
►►──keyword─▼─repeatable_argument─┴───────────────────────────────────►◄
```

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

```
             ┌─default_argument───┐
►►──keyword──┴─alternate_argument─┴──────────────────────────────────►◄
```

- 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.

```
    (1)    (2)      (3)        (4)  (5)                               (9) (10)
►►───────#──────pragma──────comment────(──┬─compiler──────────┬──────)─────────►◄
                                          ├─date──────────────┤
                                          ├─timestamp─────────┤
                                          │            (6)    │
                                          ├─copyright─┐       │
                                          └─user──────┴─┬─────┴───────────┐
                                                        │  (7)        (8) │
                                                        └─,──"─token_sequence─"─┘
```
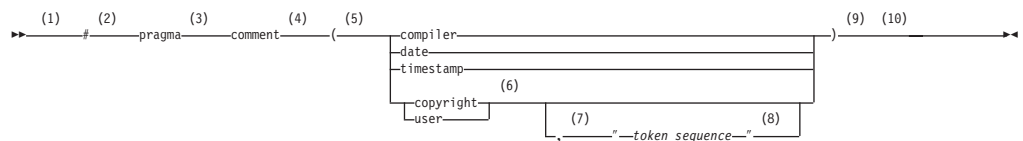
Notes:

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.

The examples for installation information are labelled as either *Example* or *Basic example*. *Basic examples* are intended to document a procedure as it would be performed during a basic, or default, installation; these need little or no modification.

Related information

The following sections provide related information for XL C:

IBM XL C information

XL C provides product information in the following formats:

- README files

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

- Installable man pages

  Man pages are provided for the compiler invocations and all command-line utilities provided with the product. Instructions for installing and accessing the man pages are provided in the *IBM XL C for AIX , V10.1 Installation Guide*.

- Information center

  The information center of searchable HTML files can be launched on a network and accessed remotely or locally. Instructions for installing and accessing the online information center are provided in the *IBM XL C for AIX , V10.1 Installation Guide*.

  The information center is viewable on the Web at http://publib.boulder.ibm.com/infocenter/comphelp/v101v121/index.jsp.

- PDF documents

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

The following files comprise the full set of XL C product information:

Table 2. XL C PDF files

Document title	PDF file name	Description
IBM XL C for AIX , V10.1 Installation Guide, GC23-8885-00	install.pdf	Contains information for installing XL C and configuring your environment for basic compilation and program execution.
Getting Started with IBM XL C for AIX , V10.1, GC23-8896-00	getstart.pdf	Contains an introduction to the XL C product, with information on setting up and configuring your environment, compiling and linking programs, and troubleshooting compilation errors.
IBM XL C for AIX , V10.1 Compiler Reference, SC23-8882-00	compiler.pdf	Contains information about the various compiler options, pragmas, macros, environment variables, and built-in functions, including those used for parallel processing.
IBM XL C for AIX , V10.1 Language Reference, SC23-8884-00	langref.pdf	Contains information about the C programming languages, as supported by IBM, including language extensions for portability and conformance to nonproprietary standards.
IBM XL C for AIX , V10.1 Optimization and Programming Guide, SC23-8883-00	proguide.pdf	Contains information on advanced programming topics, such as application porting, interlanguage calls with Fortran code, library development, application optimization and parallelization, and the XL C high-performance libraries.

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

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

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

Standards and specifications documents

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

- *Information Technology – Programming languages – C, ISO/IEC 9899:1990*, also known as *C89*.
- *Information Technology – Programming languages – C, ISO/IEC 9899:1999*, also known as *C99*.
- *Information Technology – Programming languages – Extensions for the programming language C to support new character data types, ISO/IEC DTR 19769*. This draft technical report has been accepted by the C standards committee, and is available at http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1040.pdf.
- *AltiVec Technology Programming Interface Manual*, Motorola Inc. This specification for vector data types, to support vector processing technology, is available at http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf.

- *Information Technology – Programming Languages – Extension for the programming language C to support decimal floating-point arithmetic, ISO/IEC WDTR 24732*. This draft technical report has been submitted to the C standards committee, and is available at http://www.open-std.org/JTC1/SC22/WG14/www/docs/ n1176.pdf.
- *Decimal Types for C++: Draft 4* http://www.open-std.org/jtc1/sc22/wg21/docs/ papers/2006/n1977.html

Other IBM information

- *AIX Commands Reference, Volumes 1 - 6, SC23-4888*
- *Technical Reference: Base Operating System and Extensions, Volumes 1 & 2, SC23-4913*
- *AIX National Language Support Guide and Reference*, SC23-4902
- *AIX General Programming Concepts: Writing and Debugging Programs, SC23-4896*
- *AIX Assembler Language Reference*, SC23-4923

  All AIX information is available at http://publib.boulder.ibm.com/infocenter/ pseries/v5r3/index.jsp.

- *Parallel Environment for AIX: Operation and Use*
- *ESSL for AIX V4.2 Guide and Reference*, SA22-7904, available at http://publib.boulder.ibm.com/clresctr/windows/public/esslbooks.html

Other information

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

Technical support

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

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

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

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 information or any other XL C information, send your comments by e-mail to compinfo@ca.ibm.com.

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

Chapter 1. Language levels and language extensions

The C language described in this reference is based on the standards listed in "Standards and specifications documents" on page xi.

We refer to the following language specifications as "base language levels" in order to introduce the notion of an extension to a base. In this context the base language levels refer to the following specifications:

- C99
- C89

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

In addition to the features supported by the base levels, XL C contains language extensions that enhance usability and facilitate porting programs to different platforms, including:

- "Extensions related to GNU C"
- "Extensions supporting the AltiVec Programming Interface" on page 2

You can control the language level to be used for compilation through several mechanisms, including:

- various invocation commands in the *XL C Compiler Reference*
- the **-qlanglvl** option in the *XL C Compiler Reference*

With a few exceptions, almost all of the language extensions are supported when you compile using the basic invocation command **xlc**.

The default language level for the **xlc** invocation command is **extc99**, which includes all of the features introduced by the C99 standard, and most of the IBM extensions described in this information. For a complete listing of the C extensions and various methods for enabling them, see Chapter 11, "The IBM XL C language extensions," on page 211.

For information on the various methods for controlling the language level for compilation, see Invoking the compiler in the *XL C Compiler Reference* and **-qlanglvl** in the *XL C Compiler Reference*.

Extensions to support decimal floating-point hardware

XL C supports the built-in C decimal floating-point types proposed in *Information Technology – Programming Languages – Extension for the programming language C to support decimal floating-point arithmetic, ISO/IEC WDTR 24732*. This support is enabled with the **-qdfp** in the *XL C Compiler Reference* option.

Extensions related to GNU C

Certain language extensions that correspond to GNU C features are implemented to facilitate portability. These include extensions to C89 and C99. Throughout this information, the text indicates the IBM extensions that have been implemented for

1

compatibility with GNU C ; a complete list of these is also provided in "Extensions for GNU C compatibility" on page 213.

Extensions supporting the AltiVec Programming Interface

XL C supports AltiVec vector types when vector support is enabled. These language extensions exploit the SIMD and parallel processing capabilities of the PowerPC® processor, and facilitate the associated optimization techniques. The IBM implementation of the AltiVec Programming Interface specification is an extended implementation, which, for the most part, matches the syntax and semantics of the GNU C implementation. In addition to the text provided throughout this information which describes the behavior of the vector extensions, a list of the IBM extensions to the AltiVec Programming Interface is also provided in "Extensions for vector processing support" on page 215.

Chapter 2. 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." The different types of linkages are discussed in "Program linkage" on page 7.

Related reference

"Storage class specifiers" on page 41

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. 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. Differences in terminology between C and C++

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.

Block 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

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

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

File 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.

Related reference

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[])
                {

                    printf("%d\n", i);               /* Prints 1 */

                    {
                        int i = 2, j = 3;            /* i and j defined at block scope */
                                                     /* global definition of i is hidden */
                        printf("%d\n%d\n", i, j);    /* Prints 2, 3 */

                        {
                            int i = 0;  /* i is redefined in a nested block */
                                        /* previous definitions of i are hidden */
                            printf("%d\n%d\n", i, j); /* Prints 0, 3 */
                        }

                        printf("%d\n", i);           /* Prints 2 */

                    }

                    printf("%d\n", i);               /* Prints 1 */

                    return 0;

                }
```

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. 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 these types must be unique within a single scope:
  - Enumerations
  - Structures and unions
- *Members* of structures, unions, and classes must be unique within a single structure, union, or class type.
- *Statement labels* have function scope and must be unique within a function.
- All other *ordinary identifiers* must be unique within a single scope:
  - C function names
  - 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.

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.

Related reference

"The static storage class specifier" on page 42

"The extern storage class specifier" on page 43

"Function storage class specifiers" on page 173

"Type qualifiers" on page 67

Internal linkage

The following kinds of identifiers have internal linkage:

- Objects, references, or functions explicitly declared `static`
- Objects or references declared in global scope with the specifier `const` and neither explicitly declared `extern`, nor previously declared to have external linkage
- Data members of an anonymous union

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.

Related reference

"File scope" on page 5

"The static storage class specifier" on page 42

"The static storage class specifier" on page 173

External linkage

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 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.

Related reference

"The extern storage class specifier" on page 43

"Compatibility of arrays" on page 85

"The extern storage class specifier" on page 173

No linkage

The following kinds of identifiers have no linkage:

- Names that have neither external or internal linkage
- Names declared in local scopes (with exceptions like certain entities declared with the `extern` keyword)
- Identifiers that do not represent an object or a function, including labels, enumerators, `typedef` names that refer to entities with no linkage, type names, function parameters, and template names

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

Chapter 3. Lexical Elements

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

Tokens

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

- Keywords
- Identifers
- Literals
- Punctuators and operators

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.

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.

Table 4. C keywords

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

▶ C Standard C at the C99 level also reserves the following keywords:

Table 5. C99 keywords

_Bool	_Imaginary[1]
_Complex	inline
	restrict

Note:

1. The keyword `_Imaginary` is reserved for possible future use. For complex number functionality, use `_Complex`; see Complex literals for details.

Keywords for language extensions (IBM extension)

In addition to standard language keywords, XL C reserves the following keywords for use in language extensions:

Table 6. Keywords for C language extensions

__alignof	_Decimal32[4]	__restrict
__alignof__	_Decimal64[4]	__restrict__
__asm (C only)	_Decimal128[4]	__signed__
__asm__ (C only)	__extension__	__signed
__attribute__	__label__	__volatile__
__attribute	__imag__	__thread[5]
bool	__inline__ [2]	typeof[3]
__complex__	pixel[1]	__typeof__
__const__	__pixel[1]	vector[1]
	__real__	__vector[1]

Note:

1. These keywords are recognized only in a vector declaration context, when vector support is enabled.
2. The __inline__ keyword uses the GNU C semantics for inline functions. For details, see "Linkage of inline functions" on page 174.
3. typeof is only recognized when **-qkeyword=typeof** is in effect.
4. These keywords are recognized only when **-qdfp** is enabled.
5. __thread is only recognized when **-qtls** is enabled.

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

Related reference

See -qlanglvl in the XL C Compiler Reference

See -qkeyword in the XL C Compiler Reference

"Vector types (IBM extension)" on page 52

"Identifiers"

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

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

```
>>──┬─letter─┬──┬──────┬─────────────────────────────────────><
    └───_────┘  └─┬─letter─┬┘
                  ├─digit──┤
                  └───_────┘
```

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.

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

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

Reserved identifiers

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

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

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. 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.

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.

Assembly labels (IBM extension)

The compiler binds each non-static external variable and function name in the source code to a name that it generates in the object file and any assembly code that is emitted. For compatibility with GCC, the compiler implements an extension to standard C that allows you to specify the name to be used in the object file and assembly code, by applying an assembly *label* to the declaration of a global variable or function prototype. You can also define names that do not start with an underscore even on systems where an underscore is normally prepended to the name of a function or variable.

Assembly label syntax
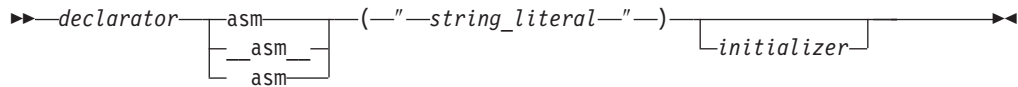
The *string_literal* is a valid assembly name that is to be bound to the given object or function. For a label applied to a function declaration, the name must specify an existing function that is defined in any compilation unit; if no definition is available, a link-time error will occur. For a label applied to a variable declaration, no other definition is required.

The following are examples of assembly label specifications:
```
void func3() __asm__("foo3");
int i __asm("abc");
char c asm("abcs") = 'a';
```

The following are restrictions on the use of assembly labels:
- Assembly labels cannot be specified on local or static variables.
- The same assembly label name cannot be applied to multiple identifiers in the same compilation unit.
- The assembly label name cannot be the same as any other global identifier name in the same compilation unit, unless the label name and identifier name are used for the same variable or function declaration.
- The assembly label cannot be specified on typedef declarations.
- An assembly label cannot be the same as a name specified on a different variable or function by a previous **#pragma map** directive. Similarly, the map

name specified by a **#pragma map** directive cannot be the same as a name specified by a previous assembly label on a different variable or function.
- You cannot apply an assembly label to an identifier that has been mapped to a different name by a **#pragma map** directive on a previous declaration of that variable or function. Similarly, you cannot specify a **#pragma map** directive on an identifier that has previously been remapped by an assembly label.
- If you apply different labels to multiple declarations of the same variable or function, the first specification is honored, and all subsequent assembly labels are ignored with a warning.

Related reference

"The Unicode standard" on page 30

"Keywords" on page 11

See -qlanglvl in the XL C Compiler Reference

"Function declarations and definitions" on page 169

See #pragma map in the XL C Compiler Reference

"The alias function attribute" on page 180

Variables in specified registers (IBM extension)

"Inline assembly statements (IBM extension)" on page 163

See -qreserved_reg in the XL C Compiler Reference

Literals

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

Every *literal* has a value and a data type. The value of any literal does not change while the program runs and must be in the range of representable values for its type. The following are the available types of literals:
- "Integer literals"
- "Boolean literals" on page 17
- "Floating-point literals" on page 17
- ▶ IBM   "Vector literals (IBM extension)" on page 22
- "Character literals" on page 24
- "String literals" on page 24

Integer literals

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 may have a prefix that specifies its base, or a suffix that specifies its type.

Integer literal syntax

```
►►──┬─decimal_constant─────┬──┬─────────────────────────┬──►◄
    ├─octal_constant───────┤  │  ┌─l──┐  ┌─u─┐          │
    └─hexadecimal_constant─┘  │  ├─L──┤  └─U─┘          │
                              │  ├─ll─┤                 │
                              │  └─LL─┘                 │
                              │  ┌─u─┐  ┌─l──┐          │
                              └──┤   │──├─L──┤          │
                                 └─U─┘  ├─ll─┤
                                        └─LL─┘
```

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

Integer literal	Possible data types
unsuffixed decimal	int, long int, long long int[1]
unsuffixed octal or hexadecimal	int, unsigned int, long int, unsigned long int, long long int[1], unsigned long long int[1]
decimal, octal, or hexadecimal suffixed by u or U	unsigned int, unsigned long int, unsigned long long int[1]
decimal suffixed by l or L	long int, long long int[1]
octal or hexadecimal suffixed by l or L	long int, unsigned long int, long long int[1], unsigned long long int[1]
decimal, octal, or hexadecimal suffixed by both u or U, and l or L	unsigned long int, unsigned long long int[1]
decimal suffixed by ll or LL	long long int
octal or hexadecimal suffixed by ll or LL	long long int, unsigned long long int
decimal, octal, or hexadecimal suffixed by both u or U, and ll or LL	unsigned long long int
Note:	
1. Assumes compilation with **xlc** or **c99** or with **-qlanglvl=stdc99** or **-qlanglvl=extc99**.	

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

```
                    ┌─────────────┐
                    │             │
►►──digit_1_to_9────▼─digit_0_to_9──┴──────────────────────►◄
```

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.

The following are examples of decimal literals:

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

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

The following are examples of hexadecimal integer literals:

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

Octal integer literals

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

Octal integer literal syntax

The following are examples of octal integer literals:

```
0
0125
034673
03245
```

Boolean literals

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

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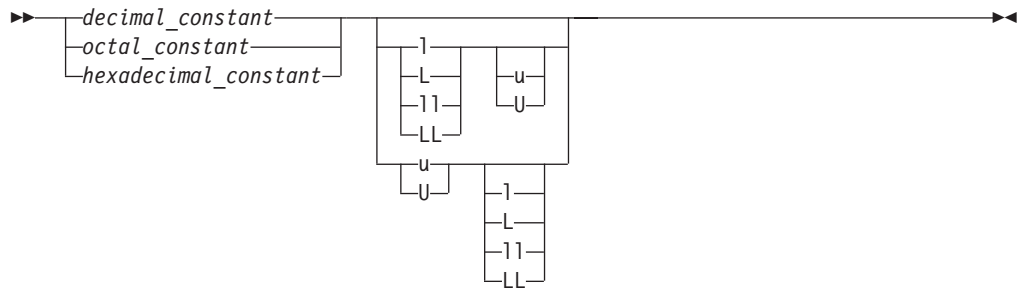
Floating-point literals are numbers that have a decimal point or an exponential part. They can be represented as:

- Real literals
  - Binary floating-point literals
  - Hexadecimal floating-point literals
  - ▷ **IBM** Decimal floating-point literals (IBM extension)
- 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
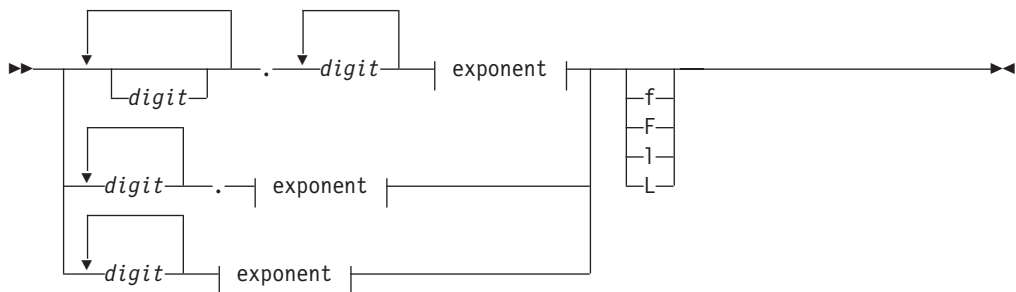
Exponent:

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

Floating-point constant	Value
1e+5	100000
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

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

- 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:

```
├──┬─p─┬──┬────┬──┬─►─digit_0_to_9─┬──────────────────────────────────┤
   └─P─┘  ├─+──┤  └───────◄────────┘
          └─-──┘
```

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.

Decimal floating-point literals (IBM extension)

A real decimal 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.

Decimal floating-point literal syntax

```
►►─┬─┬─►─────┬──.─┬─►─digit─┬──exponent──┬─┬─df─┬─►◄
   │ └─digit─┘    └────◄────┘            │ ├─DF─┤
   │                                     │ ├─dd─┤
   ├─┬─►─digit─┬──.──exponent────────────┤ ├─DD─┤
   │ └────◄────┘                         │ ├─dl─┤
   │                                     │ └─DL─┘
   └─┬─►─digit─┬──exponent───────────────┘
     └────◄────┘
```

Exponent:

```
├──┬─e─┬──┬────┬──┬─►─digit─┬──────────────────────────────────────┤
   └─E─┘  ├─+──┤  └───◄─────┘
          └─-──┘
```
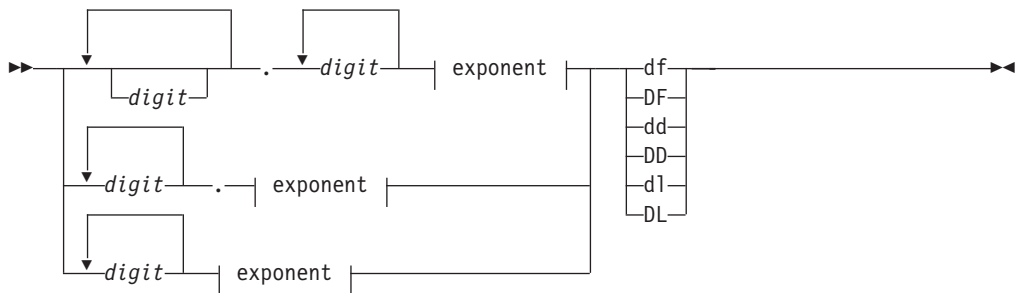
The suffix df or DF indicates a type of _Decimal32, the suffix dd or DD indicates a type of _Decimal64, and the suffix dl or DL indicates a type of _Decimal128. If a suffix is not specified, the floating-point constant has a type double.

You cannot mix cases in the literal suffix.

The following are examples of decimal floating-point literal declarations:

```
_Decimal32 a = 22.2df;
_Decimal64 b = 33.3dd;
```

Note: Decimal floating-point literal suffixes are recognized only when the **-qdfp** option is enabled.

Complex literals

Complex literals, which are a C99 feature, are constructed in two parts: the real part, and the imaginary part.

Complex literal syntax

```
►►──┤ real part ├──┬────┬──┤ imaginary part ├──────────────────────►
                   └─ + ─┘

►─┤ Imaginary part: ├──floating-point constant──*──_Complex_I───────►◄
```

Real part:
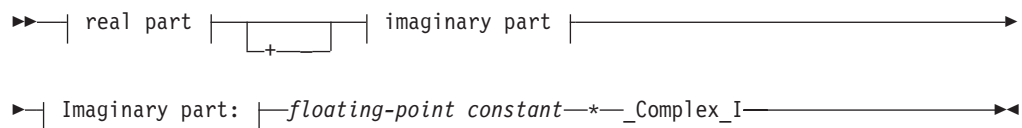
```
├──floating-point constant─────────────────────────────────────────┤
```

The *floating-point constant* can be specified as a decimal or hexadecimal floating-point constant (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*.

▶ **IBM** For ease of porting applications developed with GNU C, XL C also allows you to indicate the imaginary part of a complex literal with a suffix, in addition to the standard suffixes that indicate the type of the complex number (`float`, `double`, or `long double`).

The simplified syntax for a complex literal using the GNU suffixes is as follows:
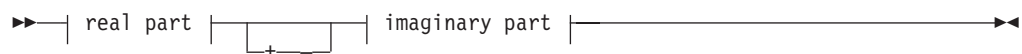
```
►►──┤ real part ├──┬────┬──┤ imaginary part ├──────────────────────►◄
                   └─ + ─┘
```
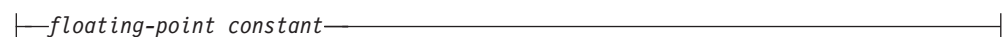
real part:

```
├──floating-point constant─────────────────────────────────────────┤
```

imaginary part:

```
├──floating-point constant──imaginary-suffix───────────────────────┤
```

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

imaginary-suffix is one of the suffixes i, I, j, or J, representing the imaginary unit.

For example, the declaration

```
varComplex = 3.0f + 4.0fi;
```

initializes the complex variable varComplex to a value of 3.0 + 4.0*i*. ◢ IBM ◀

Vector literals (IBM extension)

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

Vector literal syntax

```
>>──(──vector_type──)──(──┬─────────────────────┬──)──────────────────><
                          └─<──constant_expression──┘
                                ┌─,─┐
```
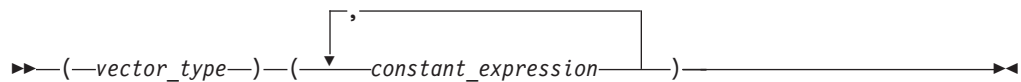
The *vector_type* is a supported vector type; see "Vector types (IBM extension)" on page 52 for a list of these.

The *constant_expression* can be either of the following:
- A single expression, which will initialize all elements of the vector to the same value
- A comma-separated list of expressions, the number of which is determined by the type of the vector. The number of constant expressions must be exactly:

  4   For vector int, vector long, and vector float types.

  8   For vector short and vector pixel types.

  16   For vector char types.

The following table shows the supported vector literals and how the compiler interprets them to determine their values.

Table 7. Vector literals

Syntax	Interpreted by the compiler as
(vector unsigned char)(*unsigned int*)	A set of 16 unsigned 8-bit quantities that all have the value of the single integer.
(vector unsigned char)(*unsigned int*, ...)	A set of 16 unsigned 8-bit quantities with the value specified by each of 16 integers.
(vector signed char)(*int*)	A set of 16 signed 8-bit quantities that all have the value of the single integer.
(vector signed char)(*int*, ...)	A set of 16 signed 8-bit quantities with the value specified by each of 16 integers.
(vector bool char)(*unsigned int*)	A set of 16 unsigned 8-bit quantities that all have the value of the single integer.

Table 7. Vector literals (continued)

Syntax	Interpreted by the compiler as
(vector bool char)(*unsigned int*, ...)	A set of 16 unsigned 8-bit quantities with a value specified by each of 16 integers.
(vector unsigned short)(*unsigned int*)	A set of 8 unsigned 16-bit quantities that all have the value of the single integer.
(vector unsigned short)(*unsigned int*, ...)	A set of 8 unsigned 16-bit quantities with a value specified by each of 8 integers.
(vector signed short)(*int*)	A set of 8 signed 16-bit quantities that all have the value of the single integer.
(vector signed short)(*int*, ...)	A set of 8 signed 16-bit quantities with a value specified by each of 8 integers.
(vector bool short)(*unsigned int*)	A set of 8 unsigned 16-bit quantities that all have the value of the single integer.
(vector bool short)(*unsigned int*, ...)	A set of 8 unsigned 16-bit quantities with a value specified by each of 8 integers.
(vector unsigned int)(*unsigned int*)	A set of 4 unsigned 32-bit quantities that all have the value of the single integer.
(vector unsigned int)(*unsigned int*, ...)	A set of 4 unsigned 32-bit quantities with a value specified by each of 4 integers.
(vector signed int)(*int*)	A set of 4 signed 32-bit quantities that all have the value of the single integer.
(vector signed int)(*int*, ...)	A set of 4 signed 32-bit quantities with a value specified by each of 4 integers.
(vector bool int)(*unsigned int*)	A set of 4 unsigned 32-bit quantities that all have the value of the single integer.
(vector bool int)(*unsigned int*, ...)	A set of 4 unsigned 32-bit quantities with a value specified by each of 4 integers.
(vector float)(*float*)	A set of 4 32-bit single-precision floating-point quantities that all have the value of the single float.
(vector float)(*float*, ...)	A set of 4 32-bit single-precision floating-point quantities with a value specified by each of 4 floats.
(vector pixel)(*unsigned int*)	A set of 8 unsigned 16-bit quantities that all have the value of the single integer.
(vector pixel)(*unsigned int*, ...)	A set of 8 unsigned 16-bit quantities with a value specified by each of 8 integers.

For example, for an unsigned integer vector type, the literal could be either of the following:

```
(vector unsigned int)(10)   /* initializes all four elements to a value of 10  */
(vector unsigned int)(14, 82, 73, 700)    /* initializes the first element
                 to 14, the second element to 82,
                 the third element to 73, and the
                 fourth element to 700   */
```

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

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*.

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.

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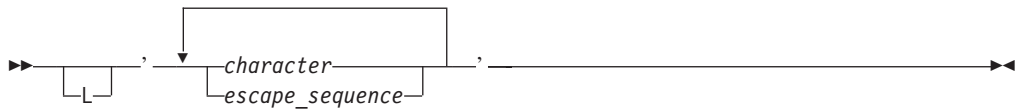
At least one character or escape sequence must appear in the character literal, and the character literal must appear on a single logical source line.

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 29 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 at the C99 language level.

The following are examples of character literals:
```
'a'
'\''
L'0'
'('
```

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*.

The type of a narrow string literal is array of char. The type of a wide string literal is array of wchar_t.

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

String literal syntax

```
>>--+---+--"--+--+-character---------+--+--"----------------------><
     |_L_|    |  |_escape_sequence_| |
              |_____<___|
```
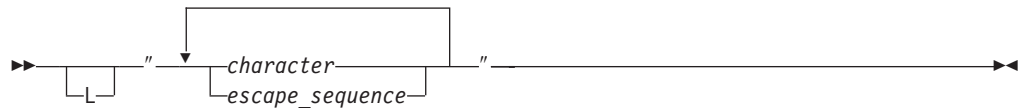
Multiple spaces contained within a string literal are retained.

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 at the C99 language level.

▶ **IBM**   The Pascal string form of a string literal is also accepted, provided that you compile with the **-qmacpstr** option.

The following are 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";
```

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 ";
```

In the following example, the string literal second causes a compile-time error.

```
char *first = "This string continues onto the next\
     line, where it ends.";                        /* compiles successfully.  */

char *second = "The comment makes the \       /* continuation symbol
     */   invisible to the compiler.";        /* compilation error.     */
```

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 writeable. By default, the compiler considers strings to be read-only. XL C may allocate only one location for a read-only string; all occurrences will refer to that one location. However, that area of storage is potentially write-protected. If strings are writeable, then each occurrence of the string will have a separate, distinct storage location that is always modifiable. You can use the **#pragma strings** directive or the **-qro** 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 will be concatenated to produce a single string. For example:

```
"hello " "there"     /* is equivalent to "hello there"  */
"hello" "there"      /* is equivalent to "hellothere"                   */
```

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

If a wide string literal and a narrow string literal are adjacent, as in the following:
`"hello " L"there"`

the result is a wide string literal.

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. 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

"Integral types" on page 47

"Integral conversions" on page 101

"Boolean types" on page 48

"Boolean conversions" on page 102

"Floating-point types" on page 49

"Floating-point conversions" on page 102

"Unary expressions" on page 115

Complex floating point types

 See -qlanglvl in the XL C Compiler Reference

"Character types" on page 50

"Source program character set" on page 28

"The Unicode standard" on page 30

String concatenation of u-literals

 See -qro in the XL C Compiler Reference

 See #pragma strings in the XL C Compiler Reference

 See #pragma map in the XL C Compiler Reference

 See -qsourcetype in the XL C Compiler Reference

 See -qmacpstr in the XL C Compiler Reference

"Literals" on page 111

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 defines the following tokens as punctuators, operators, or preprocessing tokens:

Table 8. C punctuators

[ ]	( )	{ }	,	:	;
*	=	...	#		

Table 8. C punctuators (continued)

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

Alternative tokens

The following table lists alternative representations for some operators and punctuators:

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

In addition to the operators and punctuators listed above, the C99 language level provides the following alternative representations, defined as macros in the header file `iso646.h`.

Operator or punctuator	Alternative representation
&&	and
|	bitor
||	or
^	xor
~	compl
&	bitand
&=	and_eq
|=	or_eq
^=	xor_eq
!	not
!=	not_eq

Related reference

Source program character set

The following lists 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).
  - The split vertical bar (¦) character in ASCII.

- 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.

Related reference

"Literals" on page 15

"Punctuators and operators" on page 26

Characters in identifiers

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. To instruct the compiler to recognize multibyte character sets as source input, be sure to compile with the **-qmbcs** option.

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 30

  "Character types" on page 50

  See -qmbcs in the XL C Compiler Reference

  "Comments" on page 33

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

```
►►─\─┬─escape_sequence_character─┬────────────────────────────────►◄
       ├─x─hexadecimal_digits────┤
       └─octal_digits────────────┘
```
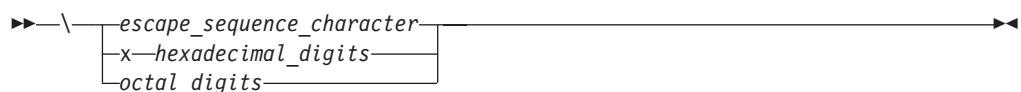
An escape sequence contains a backslash (\) symbol followed by one of the escape sequence characters or an octal or hexadecimal number. A hexadecimal escape sequence contains an x followed by one or more hexadecimal digits (0-9, A-F, a-f).

An octal escape sequence uses up to three octal digits (0-7). The value of the hexadecimal or octal number specifies the value of the desired character or wide character.

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

The escape sequences and the characters they represent are:

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

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

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

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

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

This statement results in the following output:

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

The Unicode standard

The *Unicode Standard* is the specification of an encoding scheme for written characters and text. It is a universal standard that enables consistent encoding of multilingual text and allows text data to be interchanged internationally without conflict. The ISO standard for C refers 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.

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 \uNNNN	 NNNNNNNN 0000NNNN

C99 disallows 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 C Standards Committee has approved the implementation of *u-literals* and *U-literals* to support Unicode UTF-16 and UTF-32 character literals, respectively. To enable support for UTF literals in your source code, you must compile with the option **-qutf** enabled. The following table shows the syntax for UTF literals.

Table 9. 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.

XL C implements the macros `uint_least16_t` and `uint_least32_t`, which are defined in the header file `stdint.h`, as data types for UTF-16 and UTF-32 characters. The following example defines an array of `uint_least16_t`, including the characters represented by code points 1234 and 8180:

```
#include <stdint.h>

uint_least16_t  msg[] = u"ucs characters \u1234 and \u8180 ";
```

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**
> "Identifiers" on page 12
> "Literals" on page 15
> "Punctuators and operators" on page 26
> "Multibyte characters" on page 28
>
> See -qutf in the XL C Compiler 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.

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. XL C does not replace digraphs in string literals or in character literals. For example:

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

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

> **Related reference**
> "Punctuators and operators" on page 26

See -qdigraph in the XL C Compiler Reference

Trigraph sequences

Some characters from the C character set are not available in all environments. You can enter these characters into a 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;
```

> **Related reference**
>
> "Bitwise negation operator ~" on page 117
>
> "Bitwise exclusive OR operator ^" on page 131
>
> "Bitwise inclusive OR operator |" on page 131

Comments

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

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

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

You can also include multibyte characters; to instruct the compiler to recognize multibyte characters in the source code, compile with the **-qmbcs** option.

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**
>
> See -qmbcs in the XL C Compiler Reference
>
> See -qlanglvl in the XL C Compiler Reference
>
> See -qcpluscmt in the XL C Compiler Reference
>
> "Multibyte characters" on page 28

Chapter 4. Data objects and declarations

The topics in this section 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 5, "Declarators," on page 77.

> **Related reference**
> Chapter 8, "Statements," on page 145

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*.

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 are created from the set of basic types, and include arrays, pointers, structures, unions, and enumerations.

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.

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 and structures.

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

Table 10. 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]	

Note:

1. 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."
3. The C standard does not classify enumerations as either scalar or aggregate.

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

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 84.

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

When two compatible types are combined, the result is a *composite type*. 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 47 discusses compatibility for basic and user-defined types in C.

C

Related reference

"The void type" on page 51

"Compatibility of arrays" on page 85

"Compatibility of pointers (C only)" on page 82

"Compatible functions (C only)" on page 172

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: The C99 standard no longer requires that all declarations appear at the beginning of a function before the first statement. 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 41, which specify storage duration and linkage
- "Type specifiers" on page 47, which specify data types
- "Type qualifiers" on page 67, which specify the mutability of data values
- Declarators, which introduce and include identifiers
- "Initializers" on page 86, which initialize storage with initial values

▶ **IBM** In addition, for compatibility with GCC, XL C allows you to use *attributes* to modify the properties of data objects. *Type* attributes, which can be used to modify the definition of user-defined types, are described in "Type attributes (IBM extension)" on page 72. Variable attributes, which can be used to modify the declaration of variables, are described in "Variable attributes (IBM extension)" on page 96.

All declarations have the form:

Data declaration syntax
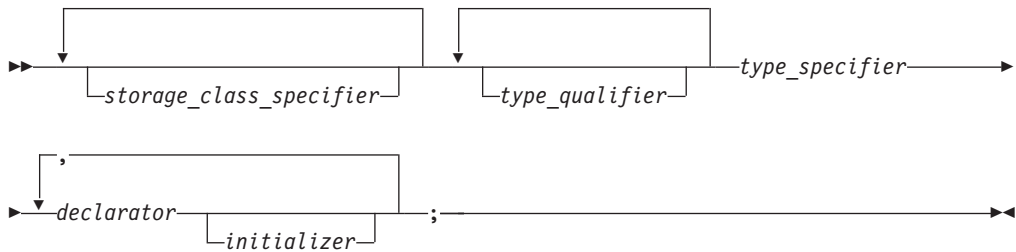
Tentative definitions

▶ **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 */
```

Related reference

"Function declarations and definitions" on page 169

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. The storage class specifiers are:

- auto
- static
- extern
- register
- ▶ IBM ◀ __thread

    Related reference

    Chapter 2, "Scope and linkage," on page 3

    "Function storage class specifiers" on page 173

    "Initializers" on page 86

    "Structures and unions" on page 55

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, memory is allocated for the object at each invocation of the block.

Linkage of automatic variables

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

Related reference

"Initialization and storage classes" on page 86

"Block statements" on page 147

"The goto statement" on page 161

"Function return values" on page 176

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.

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

The `static` storage class specifier can be applied to the following declarations:
- Data objects
- 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 178 for more information.

Linkage of static variables

A declaration of an object that contains the `static` storage class specifier and has file scope gives the identifier internal linkage. Each instance of the particular identifier therefore represents the same object within one file only. 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 static variable, it is not reinitialized to 0 on successive calls to f.

> **Related reference**
>
> "Program linkage" on page 7
>
> "The static storage class specifier" on page 173
>
> "Initialization and storage classes" on page 86
>
> "Internal linkage" on page 7
>
> "Structures and unions" on page 55
>
> "Parameter declarations" on page 177
>
> "The main() function" on page 184

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.

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

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.

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

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, memory is allocated for the variable 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 (IBM extension)

You can specify that a particular hardware register is dedicated to a variable by using an `asm` *register variable* declaration. This language extension is provided for compatibility with GNU C.

Global register variables reserve registers throughout the program; stores into the reserved register are never deleted.

Local register variables do not actually reserve the registers, except when the variables are used as input or output operands in an inline assembly statement. In this case, using the variable as an asm operand guarantees that the specified register is used for the operand and is a convenient way to control which register is used.

Register variable declaration syntax

```
►►──register──variable_declaration──┬──asm────┬──("register_specifier")────────►◄
                                     ├─__asm__─┤
                                     └─__asm───┘
```
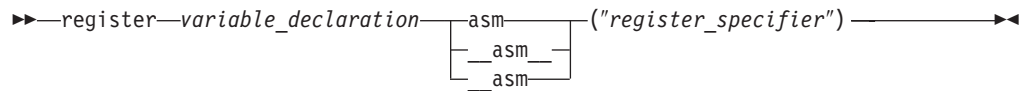
The *register_specifier* is a string representing a hardware register. The register name is CPU-specific. The following are valid register names:

r0 to r31
> General purpose registers

f0 to f31
> Floating-point registers

v0 to v31
> Vector registers (on selected processors)

The following are the rules of use for register variables:
- You cannot reserve registers for the following types of variables:
    - `long long` types
    - aggregate types
    - void types
    - `_Complex` types
    - 128-bit `long double` types
    - decimal floating-point types
- General purpose registers can only be reserved for variables of integer or pointer type.
- Floating point registers can only be reserved for variables of `float`, `double`, or 64-bit `long double` type .
- Vector registers can only be reserved for variables of vector 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.
- A register variable should not be used in an OpenMP clause or OpenMP parallel or work-sharing region.
- The register specified in the global register declaration is reserved for the declared variable only in the compilation unit in which the register declaration is specified. The register is not reserved in other compilation units unless you place the global register declaration in a common header file, or use the **-qreserved_reg** compiler option.

  Related reference

  "Initialization and storage classes" on page 86

"Block scope" on page 4

Assembly labels (IBM extension)

"Inline assembly statements (IBM extension)" on page 163

See -qreserved_reg in the XL C Compiler Reference

The __thread storage class specifier (IBM extension)

The __thread storage class marks a static variable as having *thread-local* storage duration. This means that, in a multi-threaded application, a unique instance of the variable is created for each thread that uses it, and destroyed when the thread terminates. The __thread storage class specifier can provide a convenient way of assuring thread-safety: declaring an object as per-thread allows multiple threads to access the object without the concern of race conditions, while avoiding the need for low-level programming of thread synchronization or significant program restructuring.

The **tls_model** attribute allows source-level control for the thread-local storage model used for a given variable. The **tls_model** attribute must specify one of local-exec, initial-exec, local-dynamic, or global-dynamic access method, which overrides the **-qtls** option for that variable. For example:

```
__thread int i__attribute__((tls_model("local-exec")));
```

The tls_model attribute allows the linker to check that the correct thread model has been used to build the application or shared library. The linker/loader behavior is as follows:

Table 11. Link time/runtime behavior for thread access models

Access method	Link-time diagnostic	Runtime diagnostic
local-exec	Fails if referenced symbol is imported.	Fails if module is not the main program. Fails if referenced symbol is imported (but the linker should have detected the error already).
initial-exec	None	dlopen()/load() fails if referenced symbol is not in the module loaded at execution time.
local-dynamic	Fails if referenced symbol is imported	Fails if referenced symbol is imported (but the linker should have detected the error already).
global-dynamic	None	None

Note: In order for the __thread keyword to be recognized, you must compile with the **-qtls** option. See **-qtls** in the *XL C Compiler Reference* for details.

The specifier can be applied to any of the following:
- global variables
- file-scoped static variables
- function-scoped static variables

It cannot be applied to function-scoped automatic variables .

The thread specifier can be either preceded or followed by the `static` or `extern` specifier.

```
__thread int i;
extern __thread struct state s;
static __thread char *p;
```

Variables marked with the __thread specifier can be initialized or uninitialized.
▶ C++   __thread variables must be initialized with a constant expression, and must not have a static constructor.

Applying the address-of operator (&) to a thread-local variable returns the runtime address of the current thread's instance of the variable. That thread can pass this address to any other thread; however, when the first thread terminates, any pointers to its thread-local variables become invalid.

Related reference

See -qtls in the XL C Compiler Reference

Type specifiers

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

- Fundamental or built-in types:
  - Arithmetic types
    - Integral types
    - Boolean types
    - Floating-point types
    - Character types
  - The void type
  - ▶ IBM   Vector types
- User-defined types.

  Related reference

  "Function return type specifiers" on page 176

  "typedef definitions" on page 65

  "Parameter declarations" on page 177

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

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.

Related reference

"Literals" on page 15

Integer literals

"Integral conversions" on page 101

"Arithmetic conversions and promotions" on page 101

"Enumerations" on page 62

"Integral and floating-point promotions" on page 104

"Array subscripting operator [ ]" on page 133

Boolean types

A Boolean variable can be used to hold the integer values 0 or 1, or the literals `true` or `false`, which are implicitly promoted to the integers 0 and 1 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 types are a C99 feature. To declare a Boolean variable, use the `_Bool` type specifier. ▶ **IBM** The token `bool` is recognized as a keyword in C only when used in a vector declaration context and vector support is enabled. **IBM** ◀ **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

"Literals" on page 15

"Punctuators and operators" on page 26

Boolean literals

"Boolean conversions" on page 102

"Vector types (IBM extension)" on page 52

"Integral and floating-point promotions" on page 104

"Logical negation operator !" on page 117

Floating-point types

Floating-point type specifiers fall into the following categories:
- "Real floating-point types"
- "Complex floating-point types " on page 50

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 **-qdfp** option. See **-qdfp** in the *XL C Compiler Reference* for details.
  IBM ◀

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

Table 12. Magnitude ranges of real floating-point types

Type	Range
float	approximately 1.2^{-38} to 3.4^{38}
double, long double	approximately 2.2^{-308} to 1.8^{308}
_Decimal32	0.000001^{-95} to 9.999999^{96}
_Decimal64	0.000000000000001^{-383} to 9.999999999999999^{384}
_Decimal128	$0.0000000000000000000000000000000001^{-6143}$ to $9.9999999999999999999999999999999999^{6144}$

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

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 or generic floating-point types. However, these are the restrictions on the use of decimal floating-point types with other arithmetic types:

- 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.

◀ IBM ◀

Complex floating-point types

The complex type specifiers are:
- 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).

Related reference
"Literals" on page 15
"Punctuators and operators" on page 26
Floating-point literals
"Floating-point conversions" on page 102
"Arithmetic conversions and promotions" on page 101
Complex literals (C only)
"The __real__ and __imag__ operators (IBM extension)" on page 122
"Integral and floating-point promotions" on page 104

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'`)

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

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 **-qchars** option or the **#pragma chars** directive. See **-qchars** in the *XL C Compiler Reference* for more information.

Related reference

"Literals" on page 15

"Punctuators and operators" on page 26

"Multibyte characters" on page 28

Character literals

String literals

"Arithmetic conversions and promotions" on page 101

"Integral and floating-point promotions" on page 104

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:

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

Related reference

"Overview of data objects" on page 37

"Pointers" on page 79

"Comma operator ," on page 134

"Conditional expressions" on page 136

"Function declarations and definitions" on page 169

"Conversion to void*" on page 106

"Parameter declarations" on page 177

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.

Vector types (IBM extension)

XL C supports vector processing technologies through language extensions. XL C implements the AltiVec Programming Interface specification with an extended syntax that allows type qualifiers and storage class specifiers to precede the keyword `vector` (or its alternate spelling, `__vector`) in a declaration.

Most of the legal forms of the syntax are captured in the following diagram. Some variations have been omitted from the diagram for the sake of clarity: type qualifiers such as `const` and storage class specifiers such as `static` can appear in any order within the declaration, as long as neither immediately follows the keyword `vector` (or `__vector`).

Vector declaration syntax
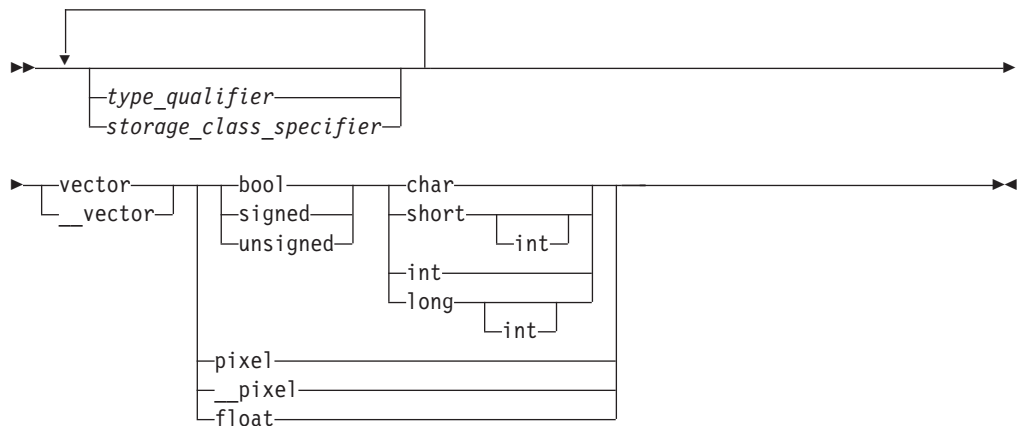
Note:

1. The keyword `vector` is recognized in a declaration context only when used as a type specifier and when vector support is enabled. The keywords `pixel`, `__pixel` and `bool` are recognized as valid type specifiers only when preceded by the keyword `vector` or `__vector`.

2. The `long` type specifier is deprecated in a vector context, and is treated as an `int`.

3. Duplicate type specifiers are ignored in a vector declaration context. In particular, `long long` is treated as `long`.

The following table lists the supported vector data types and the size and possible values for each type.

Table 13. Vector data types

Type	Interpretation of content	Range of values
vector unsigned char	16 unsigned char	0..255
vector signed char	16 signed char	-128..127
vector bool char	16 unsigned char	0, 255
vector unsigned short	8 unsigned short	0..65535
vector unsigned short int		

Table 13. Vector data types (continued)

Type	Interpretation of content	Range of values
vector signed short	8 signed short	-32768..32767
vector signed short int		
vector bool short	8 unsigned short	0, 65535
vector bool short int		
vector unsigned int	4 unsigned int	$0..2^{32}-1$
vector unsigned long		
vector unsigned long int		
vector signed int	4 signed int	$-2^{31}..2^{31}-1$
vector signed long		
vector signed long int		
vector bool int	4 unsigned int	$0, 2^{32}-1$
vector bool long		
vector bool long int		
vector float	4 float	IEEE-754 values $6.80564694 * 10^{+38}$
vector pixel	8 unsigned short	1/5/5/5 pixel

All vector types are aligned on a 16-byte boundary. An aggregate that contains one or more vector types is aligned on a 16-byte boundary, and padded, if necessary, so that each member of vector type is also 16-byte aligned.

Vector data type operators

Vector data types can use some of the unary, binary, and relational operators that are used with primitive data types. Note that all operators require compatible types as operands unless otherwise stated. These operators are not supported at global scope or for objects with static duration, and there is no constant folding.

For unary operators, each element in the vector has the operation applied to it.

Table 14. Unary operators

Operator	Integer vector types	Floating-point vector types
++	Yes	Yes
−−	Yes	Yes
+	Yes	Yes
−	Yes	Yes
~	Yes	No

For binary operators, each element has the operation applied to it with the same position element in the second operand. Binary operators also include assignment operators.

Table 15.

Operator	Integer vector types	Floating-point vector types
+	Yes	Yes

Operator	Integer vector types	Floating-point vector types
−	Yes	Yes
*	Yes	Yes
/	Yes	Yes
%	Yes	No
&	Yes	No
|	Yes	No
^	Yes	No
<<	Yes	No
>>	Yes	No
[]	Yes	Yes

Note: The [] operator returns the vector element at the position specified. If the position specified is outside of the valid range, the behavior is undefined.

For relational operators, each element has the operation applied to it with the same position element in the second operand and the results have the AND operator applied to them to get a final result of a single value.

Table 16. Relational operators

Operator	Integer vector types	Floating-point vector types
==	Yes	Yes
!=	Yes	Yes
<	Yes	Yes
>	Yes	Yes
<=	Yes	Yes
>=	Yes	Yes

For the following code:

```
vector unsigned int a = {1,2,3,4};
vector unsigned int b = {2,4,6,8};
vector unsigned int c = a + b;
int e = b > a;
int f = a[2];
vector unsigned int d = ++a;
```

c would have the value (3,6,9,12), d would have the value (2,3,4,5), e would have the value 1, and f would have the value 3.

Vector type casts

Vector types can be cast to other vector types. The cast does not perform a conversion: it preserves the 128-bit pattern, but not necessarily the value. A cast between a vector type and a scalar type is not allowed.

Vector pointers and pointers to non-vector types can be cast back and forth to each other. When a pointer to a non-vector type is cast to a vector pointer, the address should be 16-byte aligned. The referenced object of the pointer to a non-vector type

can be aligned on a sixteen-byte boundary by using either the `__align` specifier or `__attribute__((aligned(16)))`.

Related reference

"Keywords" on page 11

"Punctuators and operators" on page 26

"Boolean types" on page 48

Vector literals

"Initialization of vectors (IBM extension)" on page 89

"The __align type qualifier (IBM extension)" on page 68

"The aligned variable attribute" on page 97

"The typeof operator (IBM extension)" on page 121

User-defined types

The following are user-defined types:

- Structures and unions
- Enumerations
- Typedef definitions

Related reference

"Type attributes (IBM extension)" on page 72

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.
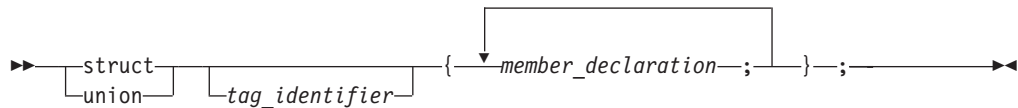
You can declare a structure or union type separately from the definition of variables of that type, as described in "Structure and union type definition" and "Structure and union variable declarations" on page 59; or you can define a structure or union data type and all variables that have that type in one statement, as described in "Structure and union type and variable definitions in a single statement" on page 60.

Structures and unions are subject to alignment considerations. For a complete discussion of alignment, see "Aligning data" in the *XL C Optimization and Programming Guide*.

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

```
>>─┬─struct─┬──┬──────────────┬──{─┬─◄──member_declaration─;─┬─┬──}─;─┬────────────►◄
   └─union──┘  └─tag_identifier─┘   └───────────────────────┘ └─────┘
```

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 60. 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
- any `void` type
- 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 element 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 for compatibility with GNU C, XL C also allows zero-extent arrays as members of structures and unions, as described in Zero-extent array members (IBM extension).

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 permitted as the last element of a structure even though it has incomplete type, provided that the structure has more than one named member. A flexible array member is a C99 feature and can be used to access a variable-length object. It is declared with an empty index, as follows:

array_identifier[ ];

For example, b is a flexible array member of Foo.
```
struct Foo{
   int a;
   int b[];
};
```

Since a flexible array member has incomplete type, you cannot apply the `sizeof` operator to a flexible array.

Any structure containing a flexible array member cannot be a member of another structure or array.

► **IBM** For compatibility with GNU C, XL C extends Standard C, to ease the restrictions on flexible arrays and allow the following:
* Flexible array members can be declared in any part of a structure, not just as the last member. The type of any member following the flexible array member is not required to be compatible with the type of the flexible array member; however, a warning is issued in this case.
* Structures containing flexible array members can be members of other structures.
* Flexible array members can be statically initialized.

In the following example:
```
struct Foo{
   int a;
   int b[];
};

struct Foo foo1 = { 55, {6, 8, 10} };
struct Foo foo2 = { 55, {15, 6, 14, 90} };
```

`foo1` creates an array b of 3 elements, which are initialized to 6, 8, and 10; while `foo2` creates an array of 4 elements, which are initialized to 15, 6, 14, and 90.

Flexible array members can only be initialized if they are contained in the outermost part of nested structures. Members of inner structures cannot be initialized. **IBM** ◄

Zero-extent array members (IBM extension)

A zero-extent array is an array with no dimensions. Like a flexible array member, a zero-extent array can be used to access a variable-length object. Unlike a flexible array member, a zero-extent array is not a C99 feature, but is provided for GNU C compatibility.

A zero-extent array must be explicitly declared with zero as its dimension:
array_identifier[0]

Like a flexible array member, 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 in this case.

Unlike a flexible array member, a structure containing a zero-extent array can be a member of another array. Also, the `sizeof` operator can be applied to a zero-extent array; the value returned is 0.

A zero-extent array can only be statically initialized with an empty set. For example:

```
struct foo{
   int a;
   char b[0];
}; bar = { 100, { } };
```

Otherwise, it must be initialized as a dyamically-allocated array.

Zero-extent array members can only be initialized if they are contained in the outermost part of nested structures. Members of inner structures cannot be initialized.

Bit field members

C allows 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

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

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 qualified and unqualified `_Bool`, `signed int`, and `unsigned int`. The default integer type for a bit field is `unsigned`.

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

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

If a series of bit fields does not add up to the size of an `int`, padding can take place. The amount of padding is determined by the alignment characteristics of the members of the structure. In some instances, bit fields can cross word boundaries.

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

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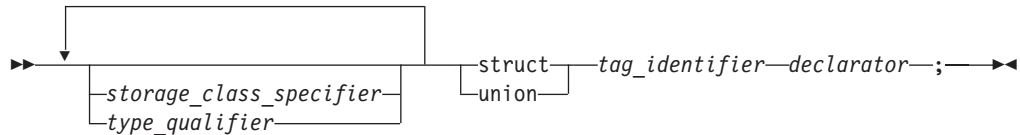
The structure `kitchen` contains eight members totalling 16 bytes. The following table describes the storage that each member occupies:

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

```
                  ┌─────────────────────────────┐
                  │            ▼                 │
►►──┬─────────────────────────────────┬──┬─struct─┬─tag_identifier─declarator─;─►◄
    │ └─storage_class_specifier─┘      │  └─union─┘
    └─type_qualifier─────────┘
```

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

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 unions

An *anonymous union* is a union without a name. It cannot be followed by a declarator. An anonymous union is not a type; it defines an unnamed object.

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
    }
```

Related reference

"The aligned type attribute" on page 73

"The packed type attribute" on page 74

"Variable length arrays" on page 84

"The aligned variable attribute" on page 97

"The __align type qualifier (IBM extension)" on page 68

"The packed variable attribute" on page 98

"Initialization of structures and unions" on page 90

"Compatibility of structures, unions, and enumerations (C only)" on page 64

"Dot operator ." on page 114

"Arrow operator ->" on page 114

"Storage class specifiers" on page 41

"Type qualifiers" on page 67

"The static storage class specifier" on page 42

"typedef definitions" on page 65

"Cast operator ()" on page 138

Enumerations

An *enumeration* is a data type consisting 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 63; 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 64.

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. C99 allows a trailing comma between the last enumerator and the closing brace.

Enumeration definition syntax

```
►►──enum──┬──────────────┬──{──▼──enumerator──┬──}──;──────────────►◄
          └─tag_identifier─┘      └─────,◄──────┘
```
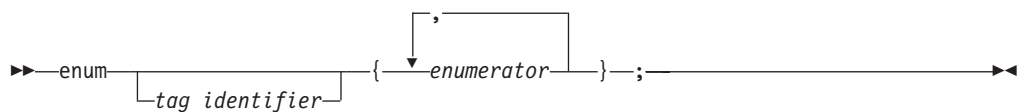
The *tag_identifier* gives a name to the enumeration type. If you do not provide a tag name, you must put all variable definitions that refer to the enumeration type within the declaration of the type, as described in "Enumeration type and variable definitions in a single statement" on page 64. Similarly, 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.

Enumeration members

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

Enumeration member declaration syntax

►►—*identifier*──►◄
        └─=—*enumeration_constant*─┘
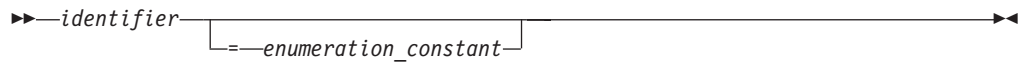
In C, an *enumeration constant* is of type int. If a constant expression is used as an initializer, the value of the expression cannot exceed the range of int (that is, INT_MIN to INT_MAX as defined in the header limits.h).

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, the leftmost enumeration constant in the list receives the value zero (0).

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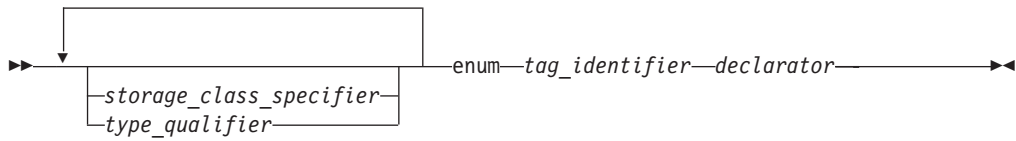
Each enumeration constant must be unique within the scope in which the enumeration is defined. In the following example, the second declarations of average and poor cause compiler errors:

```
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

```
>>─┬──────────────────────────────┬─enum──tag_identifier──declarator──────────><
   │  ┌──────────────────────────┐ │
   └─▼─┬─storage_class_specifier─┬─┘
       └─type_qualifier──────────┘
```

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

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;
```

This example 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 can not declare any additional enumeration variables using this set of enumeration constants.

> **Related reference**
>
> "Arithmetic conversions and promotions" on page 101
>
> "Integral types" on page 47
>
> "Initialization of enumerations" on page 92
>
> "Compatibility of structures, unions, and enumerations (C only)"
>
> "Integral and floating-point promotions" on page 104

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.

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 full discussion of the **-qenum** compiler option and related pragmas, see **-qenum** and **#pragma enum** in the *XL C Compiler Reference*.

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

  "Structures and unions" on page 55

  "Enumerations" on page 62

  "Arithmetic conversions and promotions" on page 101

  Structure or union type definition

  Incomplete types

typedef definitions

A `typedef` declaration lets you define your own identifiers that can be used in place of type specifiers such as `int`, `float`, and `double`. A `typedef` declaration does not reserve storage. The names you define using `typedef` are not new data types, but synonyms for the data types or combinations of data types they represent.

The namespace for a `typedef` name is the same as other identifiers. The exception to this rule is if the `typedef` name specifies a variably modified type. In this case, it has block scope.

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

▶ IBM `typedef` definitions are extended to handle vector types, provided that vector support is enabled. A vector type can be used in a `typedef` definition, and the new type name can be used in the usual ways, except for declaring other vectors. In a vector declaration context, a `typedef` name is disallowed as a type specifier. The following example illustrates a typical usage of `typedef` with vector types:

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

IBM ◀

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 following declarations are equivalent to the above declaration:

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

Similarly, `typedef` can be used to define a structure or union. For example:

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

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

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

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

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

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

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

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

Related reference

"Type names" on page 78

"Type specifiers" on page 47

"Structures and unions" on page 55

"The typeof operator (IBM extension)" on page 121

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

XL C recognizes the following type qualifiers:

- ▶ IBM `__align`
- `const`
- `restrict`
- `volatile`

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:

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

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

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: the compiler ignores duplicate type qualifiers.

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 or union, or class variable, they also apply to the members of the structure or union.

Related reference

"Program linkage" on page 7

"Structures and unions" on page 55

"Pointers" on page 79

"Overview of declarators" on page 77

"Examples of declarators" on page 78

"Assignment operators" on page 124

"Parameter declarations" on page 177

The __align type qualifier (IBM extension)

The __align qualifier is a language extension that allows you to specify an explicit alignment for an aggregate or a static (or global) variable. The specified byte boundary affects the alignment of an aggregate as a whole, not that of its members. The __align qualifier can be applied to an aggregate definition nested within another aggregate definition, but not to individual elements of an aggregate. The alignment specification is ignored for parameters and automatic variables.

A declaration takes one of the following forms:

__align qualifier syntax for simple variables

►►—*type specifier*—__align—(—*int_constant*—)—*declarator*————————————►◄

__align qualifier syntax for structures or unions

►►——__align——(—*int_constant*—)——┬─struct─┬──────────────────────►
                                   └─union──┘ └─*tag_identifier*─┘

```

►──{──*member_declaration_list*──}──;──────────────────────────────────────►◄

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

The following restrictions and limitations apply:
- The __align qualifier cannot be used where the size of the variable alignment is smaller than the size of the type alignment.
- Not all alignments may be representable in an object file.
- The __align qualifier cannot be applied to the following:
  - Individual elements within an aggregate definition.
  - Individual elements of an array.
  - Variables of incomplete type.
  - Aggregates declared but not defined.
  - Other types of declarations or definitions, such as a typedef, a function, or an enumeration.

## Examples using the __align qualifier

Applying __**align** to static or global variables:
```
int __align(1024) varA;   /* varA is aligned on a 1024-byte boundary
main()   and padded with 1020 bytes   */
{...}

static int __align(512) varB; /* varB is aligned on a 512-byte boundary
                                  and padded with 508 bytes   */

int __align(128) functionB( ); /* An error   */

typedef int __align(128) T;   /* An error   */

__align enum C {a, b, c};   /* An error   */
```

Applying __**align** to align and pad aggregate tags without affecting aggregate members:
```
__align(1024) struct structA {int i; int j;}; /* struct structA is aligned
                                                  on a 1024-byte boundary
                                                  with size including padding
                                                  of 1024 bytes   */

__align(1024) union unionA {int i; int j;}; /* union unionA is aligned
                                                  on a 1024-byte boundary
                                                  with size including padding
                                                  of 1024 bytes   */
```

Applying __**align** to a structure or union, where the size and alignment of the aggregate using the structure or union is affected:
```
__align(128) struct S {int i;};   /* sizeof(struct S) == 128   */

struct S sarray[10];   /* sarray is aligned on 128-byte boundary
                                      with sizeof(sarray) == 1280   */

struct S __align(64) svar;   /* error - alignment of variable is
                                         smaller than alignment of type */

struct S2 {struct S s1; int a;} s2; /* s2 is aligned on 128-byte boundary
                                      with sizeof(s2) == 256   */
```

Applying __**align** to an array:

```
AnyType __align(64) arrayA[10]; /* Only arrayA is aligned on a 64-byte
                                   boundary, and elements within that array
                                   are aligned according to the alignment
                                   of AnyType.  Padding is applied after the
                                   back of the array and does not affect
                                   the size of the array member itself.    */
```

Applying __**align** where the size of the variable alignment differs from the size of the type alignment:

```
__align(64) struct S {int i;};

struct S __align(32) s1;        /* error, alignment of variable is smaller
                                     than alignment of type              */

struct S __align(128) s2;       /* s2 is aligned on 128-byte boundary   */

struct S __align(16) s3[10];    /* error                                */
int __align(1) s4;              /* error                                */
__align(1) struct S {int i;};   /* error                                */
```

Related reference

"Vector types (IBM extension)" on page 52

"Structures and unions" on page 55

"The aligned variable attribute" on page 97

"The __alignof__ operator (IBM extension)" on page 119

See Aligning data in the XL C Optimization and Programming Guide

"The aligned type attribute" on page 73

"The packed type attribute" on page 74

"The packed variable attribute" on page 98

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.

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 190

The restrict type qualifier

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 is an indication to the compiler that, if the memory addressed by the restrict -qualified pointer is modified, no other pointer will access that same memory. The compiler may choose to optimize code involving restrict -qualified pointers in a way that might

otherwise result in incorrect behavior. It is the responsibility of the programmer to ensure that restrict -qualified pointers are used as they were intended to be used. Otherwise, undefined behavior may result.

If a particular chunk of memory is not modified, it can be aliased through more than one restricted pointer. The following example shows restricted pointers as parameters of foo(), and how an unmodified object can be aliased through two restricted pointers.

```
void foo(int n, int * restrict  a, int * restrict b, int * restrict  c)
{
    int i;
    for (i = 0; i < n; i++)
        a[i] = b[i] + c[i];
}
```

Assignments between restricted pointers are limited, and no distinction is made between a function call and an equivalent nested block.

```
{
    int * restrict  x;
    int * restrict  y;
    x = y; // undefined
    {
      int * restrict  x1 = x; // okay
      int * restrict  y1 = y; // okay
      x = y1;  // undefined
    }
  }
```

In nested blocks containing restricted pointers, only assignments of restricted pointers from outer to inner blocks are allowed. The exception is when the block in which the restricted pointer is declared finishes execution. At that point in the program, the value of the restricted pointer can be carried out of the block in which it was declared.

Note:

1. The restrict qualifier is represented by the following keywords (all have the same semantics):

   • The restrict keyword is recognized under compilation with **xlc** or **c99** or with the **-qlanglvl=stdc99** or **-qlanglvl=extc99** options or **-qkeyword=restrict**. The __restrict and __restrict__ keywords are recognized at all language levels.

   Related reference

   See -qlanglvl in the XL C Compiler Reference

   See -qkeyword in the XL C Compiler Reference

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 */
```

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.

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 provided to facilitate compilation of programs developed with the GNU C compiler compilers. These language features allow you to use named attributes to specify special properties of data objects. *Type* attributes apply to the definitions of user-defined types, such as structures, unions, enumerations, classes, and `typedef` definitions. Any variables 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 — aggregate types

Type attribute syntax — typedef declarations

```
►►──__attribute__──((──────────────────────────────────────))──;──────────────►◄
                      │  ┌─,──────────────────────────────┐  │
                      │  ▼                                │  │
                      └───attribute name──__attribute name__─┘
```

The *attribute name* can be specified with or without double underscore characters leading and trailing; however, using the double underscore reduces the likelihood of a name conflict with a macro of the same name. For unsupported attribute names, the XL 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 aligned type attribute"
- "The packed type attribute" on page 74
- "The transparent_union type attribute (C only)" on page 74

  Related reference

  "User-defined types" on page 55

  "Variable attributes (IBM extension)" on page 96

  "Function attributes (IBM extension)" on page 179

The aligned type attribute

The `aligned` type attribute allows you to override the default alignment mode to specify a minimum alignment value, expressed as a number of bytes, for a structure, , union, enumeration, or other user-defined type created in a `typedef` declaration. The `aligned` attribute is typically used to increase the alignment of any variables declared of the type to which the attribute applies.

aligned type attribute syntax

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

The *alignment_factor* is the number of bytes, specified as a constant expression that evaluates to a positive power of 2. You can specify a value up to a maximum 1048576 bytes. If you omit the alignment factor (and its enclosing parentheses), the compiler automatically uses 16 bytes. If you specify an alignment factor greater than the maximum, the attribute specification is ignored, and the compiler simply uses the default alignment in effect.

The alignment value that you specify will be applied to all instances of the type. Also, the alignment value applies to the variable as a whole; if the variable is an aggregate, the alignment value applies to the aggregate as a whole, not to the individual members of the aggregate.

In all of the following examples, the `aligned` attribute is applied to the structure type A. Because a is declared as a variable of type A, it will also receive the alignment specification, as will any other instances declared of type A.

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

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

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

  Related reference

See Aligning data in the XL C Optimization and Programming Guide

The packed type attribute

The packed type attribute specifies that the minimum alignment should be used for the members of a structure, union, or enumeration type. For structure or union types, the alignment is one byte for a member and one bit for a bit field member. For enumeration types, the alignment is the smallest size that will accomodate the range of values in the enumeration. All members of all instances of that type will use the minimum alignment.

packed type attribute syntax

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

Unlike the aligned type attribute, the packed type attribute is not allowed in a typedef declaration.

See Aligning data in the XL C Optimization and Programming Guide

The transparent_union type attribute (C only)

The transparent_union attribute applied to a union definition or a union typedef definition indicates the union can be used as a *transparent union*. Whenever a transparent union is the type of a function parameter and that function is called, the transparent union can accept an argument of any type that matches that of one of its members without an explicit cast. Arguments to this function parameter are passed to the transparent union, using the calling convention of the first member of the union type. Because of this, all members of the union must have the same machine representation. Transparent unions are useful in library functions that use multiple interfaces to resolve issues of compatibility.

transparent_union type attribute syntax

```
►►──__attribute__──((──┬──transparent_union───────┬──))──────────────────────────►◄
                       └──__transparent_union__──┘
```

The union must be a complete union type. The transparent_union type attribute can be applied to anonymous unions with tag names.

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

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

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

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

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

Floating-point types (`float`, `double`, `float _Complex`, or `double _Complex`) types or vector types can be members of a transparent union, but they cannot be the first member. The restriction that all members of the transparent union have the same machine representation as the first member still applies.

Related reference

"Function argument conversions" on page 107

"Cast operator ()" on page 138

Chapter 5. Declarators

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

Overview of declarators

A *declarator* designates a data object or function. A declarator can also include an initialization. Declarators appear in most data definitions and declarations and in some type definitions.

For data declarations, a declarator has the form:

Declarator syntax

Direct declarator:

Pointer operator (C only):

Declarator name (C only):

The *type_qualifiers* represent one or a combination of `const` and `volatile`.

Initializers are discussed in "Initializers" on page 86.

The following are known as *derived declarator* types, and are therefore discussed in this section:
- "Pointers" on page 79
- "Arrays" on page 83

▷ IBM In addition, for compatibility with GNU C, XL 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 this section, in "Variable attributes (IBM extension)" on page 96.

Related reference

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.

Related reference

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 17. 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 */
```

Related reference

"typedef definitions" on page 65

"Operator precedence and associativity" on page 141

"Examples of expressions and precedence" on page 143

"Parenthesized expressions ( )" on page 112

"The sizeof operator" on page 120

"The typeof operator (IBM extension)" on page 121

"Cast operator ()" on page 138

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.
• To access an array of characters as a string.

- To pass the address of a variable to a function. By referencing a variable through its address, a function can change the contents of that variable.

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 18. 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`

Related reference

"The void type" on page 51

"Type qualifiers" on page 67

"Initialization of pointers" on page 92

"Compatibility of pointers (C only)" on page 82

"Pointer conversions" on page 105

"Address operator &" on page 118

"Indirection operator *" on page 118

"Pointers to functions" on page 187

"Arrow operator ->" on page 114

"Assignment operators" on page 124

"Array subscripting operator [ ]" on page 133

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.

▶ IBM   Pointer arithmetic is defined for pointer to vector types. Given:

```
vector unsigned int *v;
```

the expression v + 1 represents a pointer to the vector following v.

Related reference

"Increment operator ++" on page 115

"Arrays" on page 83

"Decrement operator --" on page 116

Chapter 7, "Expressions and operators," on page 109

"Unary expressions" on page 115

"Addition operator +" on page 127

"Subtraction operator -" on page 127

"Array subscripting operator [ ]" on page 133

Type-based aliasing

The compiler follows the type-based aliasing rule in the C standard when the **-qalias=ansi** 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]The common coding practice of casting a

---

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,

pointer to an incompatible type and then dereferencing it violates this rule. (Note that `char` pointers are an exception to this rule.)

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 a int ptr     */
    f += 1.0;                /* compiler may discard this 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;` is never used subsequently. Thus, the optimizer may discard the statement from the generated code. If you compile the above example with optimization enabled, the `printf` statement may output `0` (zero).

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:

```
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 */
```

Related reference

- 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

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 language provides 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.

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.

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

```
>>──┬──[──constant_expression──]──┬──────────────────────><
    └─────────────◄───────────────┘
```

The *constant_expression* is a constant integer expression, indicating the size of the array, which must be positive.

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 84.

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:

```
|                |              |
|_____|_____|
 _____
↑                ↑              ↑
|                |              |
roster[0][0]     roster[0][1]
roster[1][0]
```

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

"Pointer arithmetic" on page 80

"Array subscripting operator [ ]" on page 133

"Initialization of arrays" on page 93

"Pointer conversions" on page 105

"Lvalues and rvalues" on page 109

"Indirection operator *" on page 118

"The sizeof operator" on page 120

"Parameter declarations" on page 177

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.

Variable length array declarator syntax

```
►►──array_identifier──[──┬──expression────────────┬──]────────────────►◄
                         └──type-qualifiers──┘  *
```

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. A variable length array cannot be initialized.

A variable length array can be the operand of a `sizeof` expression. In this case, the operand is evaluated at run time, and the size is neither an integer constant nor a constant expression, even though the size of each instance of a variable array does not change during its lifetime.

A variable length array can be used in a `typedef` 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.

Related reference

"Structures and unions" on page 55

Flexible array members (C only)

Compatibility of arrays

Two array types that are similarly qualified are compatible if the types of their elements are compatible. For example,

```
char ex1[25];
const char ex2[25];
```

are not compatible.

The composite type of two compatible array types is an array with the composite element type. The sizes of both original types must be equivalent if they are known. If the size of only one of the original array types is known, then the composite type has that size. For example:

```
char ex3[];
char ex4[42];
```

The composite type of `ex3` and `ex4` is `char[42]`. If one of the original types is a variable length array, the composite type is that type.

Related reference

"Overview of data objects" on page 37

"External linkage" on page 8

Initializers

An *initializer* is an optional part of a data declaration that specifies an initial value of a data object. The initializers that are legal for a particular declaration depend on the type and storage class of the object to be initialized.

The 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.

"Initialization and storage classes" discusses the rules for initialization according to the storage class of variables.

"Designated initializers for aggregate types (C only)" on page 87 describes designated initializers, which are a C99 feature that can be used to initialize arrays, structures, and unions.

The following sections discuss initialization for derived types:
- "Initialization of vectors (IBM extension)" on page 89
- "Initialization of structures and unions" on page 90
- "Initialization of pointers" on page 92
- "Initialization of arrays" on page 93

    Related reference

    "Storage class specifiers" on page 41

Initialization and storage classes
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 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.

Initialization of static variables

You initialize a static object with a constant expression, or an expression that reduces to the address of a previously declared extern or static object, possibly

modified by a constant expression. If you do not explicitly initialize a `static` (or external) variable, it will have a value of zero of the appropriate type, unless it is a pointer, in which case it will be initialized to `NULL`.

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.

Initialization of external variables

You can initialize any object with the `extern` storage class specifier at global scope. The initializer for an `extern` object must either:

- Appear as part of the definition and the initial value must be described by a constant expression; or
- Reduce to the address of a previously declared object with static storage duration. You may modify this object with pointer arithmetic. (In other words, you may modify the object by adding or subtracting an integral constant expression.)

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

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 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 41

"The static storage class specifier" on page 42

"The extern storage class specifier" on page 43

"The register storage class specifier" on page 44

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

```
►►──{──┬──.──member──=──expression──┬──}──────────────────────►◄
        └──────────,◄──────────────┘
```

Designator list syntax for arrays

```
►►──{──┬──[──array subscript──]──┬──=──expression──┬──}──────►◄
        └─────────────,◄─────────────────────────┘
```

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" on page 90
>
> "Initialization of arrays" on page 93

Initialization of vectors (IBM extension)

A vector type is initialized by a vector literal or any expression having the same
vector type. For example:

```
vector unsigned int v1;
vector unsigned int v2 = (vector unsigned int)(10);
v1 = v2;
```

XL C extends the AltiVec specification to allow a vector type to be initialized by an
initializer list. This feature is an extension for compatibility with GNU C.

Vector initializer list syntax

►►—*vector_type*——*identifier*—=—{—┬─*initializer*─┬—}—;——————————►◄
                                              └─────,────┘

The number of values in a braced initializer list must be less than or equal to the
number of elements of the vector type. Any uninitialized element will be initialized
to zero.

The following are examples of vector initialization using initializer lists:

```
vector unsigned int v1 = {1};  // initialize the first 4 bytes of v1 with 1
                               //  and the remaining 12 bytes with zeros

vector unsigned int v2 = {1,2};  // initialize the first 8 bytes of v2 with 1 and 2
                                 // and the remaining 8 bytes with zeros

vector unsigned int v3 = {1,2,3,4};  // equivalent to the vector literal
                                     // (vector unsigned int) (1,2,3,4)
```

Unlike vector literals, the values in the initializer list do not have to be constant expressions unless the initialized vector variable has static duration. Thus, the following is legal:

```
int i=1;
int foo() { return 2; }
int main()
{
   vector unsigned int v1 = {i, foo()};
   return 0;
}
```

   Related reference

   "Punctuators and operators" on page 26

   "Vector types (IBM extension)" on page 52

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 allows the initializer for an automatic member variable of a union or structure type to be a constant or non-constant expression.

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.
- 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 87.

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 };
```

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 a structure or union; the initial value of uninitialized structure members depends on the storage class associated with the structure or union variable. In a structure declared as static, any members that are not initialized are implicitly initialized to zero of the appropriate type; the members of a structure with automatic storage have no default initialization. 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.

The following definition shows a partially initialized structure:

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

The values of `temp_address` are:

Member	Value
temp_address.street_no	44
temp_address.street_name	address of string "Knyvet Ave."
temp_address.city	address of string "Hamilton"
temp_address.prov	address of string "Ontario"

Member	Value
temp_address.postal_code	Depends on the storage class of the temp_address variable; if it is static, the value would be NULL.

▶ 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" };
```

Related reference

"Structures and unions" on page 55

"Designated initializers for aggregate types (C only)" on page 87

Structure and union variable declarations

"Assignment operators" on page 124

Initialization of enumerations

The initializer for an enumeration variable contains the = symbol followed by an expression *enumeration_constant*.

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

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

Related reference

"Enumerations" on page 62

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.

Related reference

"Pointers" on page 79

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.
- 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 87.

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";
```

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

Variable attributes (IBM extension)

Variable attributes are language extensions provided to facilitate the compilation of programs developed with the GNU C compiler compilers. These language features allow you to use named attributes to specify special properties of data objects. *Variable* attributes apply to the declarations of simple variables, aggregates, and member variables of aggregates.

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

```
►►─declarator─__attribute__─((─┬──►──┬─attribute name──┬──┬─))─────────────►◄
                               │  ▲ └─__attribute name__─┘  │
                               └─────────────,──────────────┘
```

Variable attribute syntax: pre-declarator

```
►►─type specifier─__attribute__─((─┬──►──┬─attribute name──┬──┬─))───────────►
                                   │  ▲ └─__attribute name__─┘  │
                                   └─────────────,──────────────┘
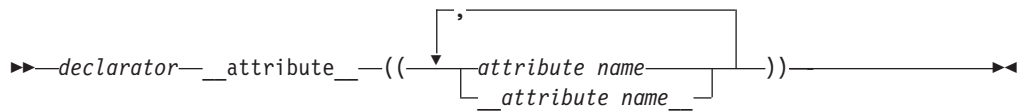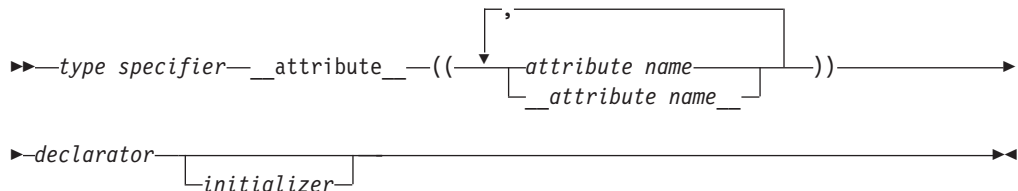
►─declarator─┬───────────┬───────────────────────────────────────────────────►◄
             └─initializer─┘
```

The *attribute name* can be specified with or without leading and trailing double underscore characters; however, using the double underscore reduces the likelihood of a name conflict with a macro of the same name. For unsupported attribute names, the XL 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:

Related reference

"Type attributes (IBM extension)" on page 72

"Function attributes (IBM extension)" on page 179

The aligned variable attribute

The `aligned` variable attribute allows you to override the default alignment mode to specify a minimum alignment value, expressed as a number of bytes, for any of the following:

- a non-aggregate variable
- an aggregate variable (such as a structure or union)
- selected member variables

The attribute is typically used to increase the alignment of the given variable.

aligned variable attribute syntax

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

The *alignment_factor* is the number of bytes, specified as a constant expression that evaluates to a positive power of 2. You can specify a value up to a maximum of 1048576 bytes. If you omit the alignment factor, and its enclosing parentheses, the compiler automatically uses 16 bytes. If you specify an alignment factor greater than the maximum, the attribute specification is ignored, and the compiler simply uses the default alignment in effect.

When you apply the `aligned` attribute to a bit field structure member variable, 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.

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  ;
```

Related reference

"Vector types (IBM extension)" on page 52

"Structures and unions" on page 55

"The __align type qualifier (IBM extension)" on page 68

"The aligned type attribute" on page 73

See Aligning data in the XL C Optimization and Programming Guide

"The __alignof__ operator (IBM extension)" on page 119

The mode variable attribute

The variable attribute mode allows you to override the type specifier in a variable declaration, to specify the size of a particular integral type.

mode variable attribute syntax

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

The valid argument for the mode is any of the of the following type specifiers that indicates a specific width:

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

The packed variable attribute

The variable attribute packed allows you to override the default alignment mode, to reduce the alignment for all members of an aggregate, or selected members of an aggregate to the smallest possible alignment: one byte for a member and one bit for a bit field member.

packed variable attribute syntax

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

Related reference

"Structures and unions" on page 55

"The packed type attribute" on page 74

"The __align type qualifier (IBM extension)" on page 68

See Aligning data in the XL C Optimization and Programming Guide

"The __alignof__ operator (IBM extension)" on page 119

The tls_model attribute

The **tls_model** attribute allows source-level control for the thread-local storage model used for a given variable. The **tls_model** attribute must specify one of `local-exec`, `initial-exec`, `local-dynamic`, or `global-dynamic` access method, which overrides the **-qtls** option for that variable. For example:

```
__thread int i__attribute__((tls_model("local-exec")));
```

The tls_model attribute allows the linker to check that the correct thread model has been used to build the application or shared library. The linker/loader behavior is as follows:

Table 19. Link time/runtime behavior for thread access models

Access method	Link-time diagnostic	Runtime diagnostic
`local-exec`	Fails if referenced symbol is imported.	Fails if module is not the main program. Fails if referenced symbol is imported (but the linker should have detected the error already).
`initial-exec`	None	`dlopen()`/`load()` fails if referenced symbol is not in the module loaded at execution time.
`local-dynamic`	Fails if referenced symbol is imported	Fails if referenced symbol is imported (but the linker should have detected the error already).
`global-dynamic`	None	None

The weak variable attribute

The `weak` variable attribute causes the symbol resulting from the variable declaration to appear in the object file as a weak symbol, rather than a global one. The language feature provides the programmer writing library functions with a way to allow variable definitions in user code to override the library declaration without causing duplicate name errors.

weak variable attribute syntax

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

Related reference

📄 See #pragma weak in the XL C Compiler Reference

"The weak function attribute" on page 184

"The alias function attribute" on page 180

Chapter 6. Type conversions

An expression of a given type is *implicitly converted* in the following situations:
- The expression is used as an operand of an arithmetic or logical operation.
- The expression is used 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).
- The expression is used in a `switch` statement. The expression will be converted to an integral type.
- The expression is used as an initialization. This includes the following:
  - An assignment is made to an lvalue that has a different type than the assigned value.
  - A function is provided an argument value that has a different type than the parameter.
  - The value specified in the `return` statement of a function has a different type from the defined return type for the function.

You can perform *explicit* type conversions using a *cast* expression, as described in "Cast expressions" on page 138.

Related reference

"The switch statement" on page 150

"The if statement" on page 149

"The return statement" on page 160

Arithmetic conversions and promotions

The following sections discuss the rules for the standard conversions for arithmetic types:
- "Integral conversions"
- "Floating-point conversions" on page 102
- "Boolean conversions" on page 102

If two operands in an expression have different types, they are subject to the rules of the *usual arithmetic conversions*, as described in "Integral and floating-point promotions" on page 104.

Related reference

"Integral types" on page 47

"Floating-point types" on page 49

"Character types" on page 50

"Enumerations" on page 62

"Compatibility of structures, unions, and enumerations (C only)" on page 64

"Unary expressions" on page 115

"Binary expressions" on page 123

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
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`.

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.

Related reference
"Literals" on page 15
"Integral types" on page 47

Boolean conversions

Boolean to integer
▶ C If the Boolean value is 0, the result is an `int` with a value of 0. If the Boolean value is 1, the result is an `int` with a value of 1.

Scalar to Boolean
▶ C If the scalar value is equal to 0, the Boolean value is 0; otherwise the Boolean value is 1.

Related reference
"Literals" on page 15
"Punctuators and operators" on page 26
"Boolean types" on page 48

Floating-point conversions

The standard rule for converting between real floating-point types (binary to binary, decimal to decimal and decimal to binary) 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 dependant on the rounding mode.

Integer to floating-point (binary or decimal)
> 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 (binary or decimal) 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.

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.

Complex to real (decimal)
> The imaginary part of the complex value is discarded. The value of the real part is converted from binary to decimal floating-point, 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.

Real (decimal) to complex
> The source value is converted from decimal to binary floating-point, according to the standard conversion rule given above, and used as the real part of the complex value. The value of the imaginary part is zero.

> **Related reference**

> "Literals" on page 15

> "Punctuators and operators" on page 26

> "Floating-point types" on page 49

Integral and floating-point promotions

When different arithmetic types are used as operands in certain types of expressions, standard conversions known as *usual arithmetic conversions* are applied. These conversions are applied according to the rank of the arithmetic type: the operand with a type of lower rank is converted to the type of the operand with a higher rank. This is known as integral or floating point *promotion*.

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 7, "Expressions and operators," on page 109 provides a list of the operators and expressions that participate in the usual arithmetic conversions.

The ranking of arithmetic types, listed from highest to lowest, is as follows:

Table 20. Conversion rankings for floating-point types

Operand type
`long double` or `long double _Complex`
`double` or `double _Complex`
`float` or `float _Complex`

Table 21. Conversion rankings for decimal floating-point types

Operand type
> IBM _Decimal128
> IBM _Decimal64
> IBM _Decimal32

Table 22. Conversion rankings for integer types

Operand type
`unsigned long long` or `unsigned long long int`
`long long` or `long long int`[1]
`unsigned long int`[1]
`long int`[2]
`unsigned int`[2]
int and enumerated types
`short int`
`char, signed char` and `unsigned char`
Boolean

Note:

1. If one operand has `unsigned long int` type and the other operand has `long long int` (or `long long`) type but the value of the `unsigned long int` cannot be represented in a `long long int` (or `long long`), both operands are converted to `unsigned long long int` (or `unsigned long long`).

2. If one operand has `unsigned int` type and the other operand has `long int` type but the value of the `unsigned int` cannot be represented in a `long int`, both operands are converted to `unsigned long int`.

Related reference

"Integral types" on page 47

"Boolean types" on page 48

"Floating-point types" on page 49

"Character types" on page 50

"Enumerations" on page 62

"Binary expressions" on page 123

"Function argument conversions" on page 107

Lvalue-to-rvalue conversions

If an lvalue appears in a situation in which the compiler expects an rvalue, the compiler converts the lvalue to an rvalue. The following table lists exceptions to this:

Situation before conversion	Resulting behavior
The lvalue is a function type.	
The lvalue is an array.	
The type of the lvalue is an incomplete type.	compile-time error
The lvalue refers to an uninitialized object.	undefined behavior
The lvalue refers to an object not of the type of the rvalue, nor of a type derived from the type of the rvalue.	undefined behavior

Related reference

"Lvalues and rvalues" on page 109

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 qualifiers as the right operand. An object pointer may be an incomplete type if the other pointer has type `void*`. ▌ C ◄

Zero constant to null pointer

A constant expression that evaluates to zero is a *null pointer constant*. This expression can be converted to a pointer. This pointer will be a null pointer (pointer with a zero value), and is guaranteed not to point to any object.

Array to pointer

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. A conversion cannot be performed if the expression is used as the operand of the & (address) operator or the `sizeof` operator.

Function to pointer

An lvalue that is a function can be converted to an 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.

Related reference

"Pointers" on page 79

"Integer constant expressions" on page 111

"Arrays" on page 83

"Pointers to functions" on page 187

"Addition operator +" on page 127

"Subtraction operator -" on page 127

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*)`

Related reference

"The void type" on page 51

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:

- Integral and floating-point values are promoted.
- Arrays or functions are converted to pointers.

  Related reference

  "Integral and floating-point promotions" on page 104

  "The transparent_union type attribute (C only)" on page 74

  "Function call expressions" on page 113

  "Function calls" on page 185

Chapter 7. 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 141 provides tables listing the precedence of all the operators described in the various sections listed above.

Related reference

"Punctuators and operators" on page 26

"Pointer arithmetic" on page 80

"Expression statements" on page 147

Lvalues and rvalues

An *object* is a region of storage that can be examined and stored into. An *lvalue* is an expression that refers to such an object. An lvalue does not necessarily permit modification of the object it designates. For example, a `const` object is an lvalue that cannot be modified. The term *modifiable lvalue* is used to emphasize that the lvalue allows the designated object to be changed as well as examined. The following object types are lvalues, but not modifiable lvalues:

- An array type
- An incomplete type
- A `const`-qualified type
- 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.

The term *rvalue* refers to a data value that is stored at some address in memory. An rvalue is an expression that cannot have a value assigned to it. Both a literal constant and a variable can serve as an rvalue. When an lvalue appears in a context that requires an rvalue, the lvalue is implicitly converted to an rvalue. The reverse, however, is not true: an rvalue cannot be converted to an lvalue. Rvalues always have complete types or the void type.

C 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.

Certain operators require lvalues for some of their operands. The table below lists these operators and additional constraints on their usage.

Operator	Requirement
& (unary)	Operand must be an lvalue.
++ --	Operand must be an lvalue. This applies to both prefix and postfix forms.

Operator	Requirement
= += -= *= %= <<= >>= &= ^= |=	Left operand must be an lvalue.

For example, all assignment operators evaluate their right operand and assign that value to their left operand. The left operand must be a modifiable lvalue or a reference to a modifiable object.

The address operator (&) requires an lvalue as an operand while the increment (++) and the decrement (--) operators require a modifiable lvalue as an operand. The following example shows expressions and their corresponding lvalues.

Expression	Lvalue
x = 42	x
*ptr = newvalue	*ptr
a++	a

> IBM When compiled with the GNU C language extensions enabled, compound expressions, conditional expressions, and casts are allowed as lvalues, provided that their operands are lvalues.

A compound expression can be assigned if the last expression in the sequence is an lvalue. The following expressions are equivalent:

```
(x + 1, y) *= 42;
x + 1, (y *=42);
```

The address operator can be applied to a compound expression, provided the last expression in the sequence is an lvalue. The following expressions are equivalent:

```
&(x + 1, y);
x + 1, &y;
```

A conditional expression can be a valid lvalue if its type is not void and both of its branches for true and false are valid lvalues. Casts are valid lvalues if the operand is an lvalue. The primary restriction is that you cannot take the address of an lvalue cast. IBM ◄

Related reference

"Lvalue-to-rvalue conversions" on page 105

"Arrays" on page 83

"Unary expressions" on page 115

"Binary expressions" on page 123

"Assignment operators" on page 124

"Array subscripting operator [ ]" on page 133

"Cast operator ()" on page 138

Primary expressions

Primary expressions fall into the following general categories:

- Names (identifiers)
- Literals (constants)
- Integer constant expressions

- Parenthesized expressions ( )

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.	no
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.	no

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 15

Integer constant expressions

An *integer compile-time constant* is a value that is determined during compilation and cannot be changed at run time. An *integer compile-time constant expression* is an expression that is composed of constants and evaluated to a constant.

An integer constant expression is an expression that is composed of only the following:

- literals

- enumerators
- `const` variables
- `static` data members of integral or enumeration types
- casts to integral types
- `sizeof` expressions, where the operand is not a variable length array

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.)

Related reference

"Pointer conversions" on page 105

"The sizeof operator" on page 120

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:

expression

*

parenthesized expression

expression

+

expression   function call

parameters

unary minus

expression   expression

− discount   *   (item   +   handling   (weight   ,   zipcode   )   )

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

"Type names" on page 78

"Operator precedence and associativity" on page 141

"The #define directive" on page 190

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 or the type void. A function call expression is always an rvalue.

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 107

"Function calls" on page 185

Member expressions

Member expressions indicate members of structures, or unions. The member operators are:

- Dot operator .
- Arrow operator ->

Dot operator .

The `.` (dot) operator is used to access structure, or union members. The member is specified by a postfix expression, followed by a `.` (dot) operator, followed by a possibly qualified identifier . The postfix expression must be an object of type `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

"Structures and unions" on page 55

Access to structure and union members

Arrow operator ->

The `->` (arrow) operator is used to access structure or union members using a pointer. A postfix expression, followed by an `->` (arrow) operator, followed by a possibly qualified identifier, designates a member of the object to which the pointer points. The postfix expression must be a pointer to an object of type `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

"Structures and unions" on page 55

"Pointers" on page 79

Access to structure and union members

Unary expressions

A *unary expression* contains one operand and a unary operator.

The supported unary operators are:
- "Increment operator ++"
- "Decrement operator --" on page 116
- "Unary plus operator +" on page 117
- "Unary minus operator -" on page 117
- "Logical negation operator !" on page 117
- "Bitwise negation operator ~" on page 117
- "Address operator &" on page 118
- "Indirection operator *" on page 118
- ▶ IBM alignof
- sizeof
- ▶ IBM typeof
- ▶ IBM __real__ and __imag__
- ▶ IBM vec_step

All unary operators have the same precedence and have right-to-left associativity, as shown in Table 28 on page 141.

As indicated in the descriptions of the operators, the usual arithmetic conversions are performed on the operands of most unary expressions.

> **Related reference**
>
> "Literals" on page 15
>
> "Punctuators and operators" on page 26
>
> "Pointer arithmetic" on page 80
>
> "Lvalues and rvalues" on page 109
>
> "Arithmetic conversions and promotions" on page 101

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. For example:

```
play = ++play1 + play2++;
```

is similar to the following expressions; `play2` is altered before `play`:

```
int temp, temp1, temp2;

temp1 = play1 + 1;
temp2 = play2;
```

```
play1 = temp1;
temp = temp1 + temp2;
play2 = play2 + 1;
play = temp;
```

The result has the same type as the operand after integral promotion.

The usual arithmetic conversions on the operand are performed.

▶ IBM ▬ 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.

Related reference

"Pointer arithmetic" on page 80

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.

For example:

```
play = --play1 + play2--;
```

is similar to the following expressions; play2 is altered before play:

```
int temp, temp1, temp2;

temp1 = play1 - 1;
temp2 = play2;
play1 = temp1;
temp = temp1 + temp2;
play2 = play2 - 1;
play = temp;
```

The result has the same type as the operand after integral promotion, but is not an lvalue.

The usual arithmetic conversions are performed on the operand.

▶ IBM ▬ 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.

Related reference

"Pointer arithmetic" on page 80

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).

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.

The following two expressions are equivalent:
```
!right;
right == 0;
```
> **Related reference**
> "Boolean types" on page 48

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** 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.

Related reference

"Trigraph sequences" on page 33

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, nor can it have the storage class `register`.

If the operand is an lvalue or function, the resulting type is a pointer to the expression type. For example, if the expression has type `int`, the result is a pointer to an object having type `int`.

If the operand is a qualified name and the member is not static, the result is a pointer to a member of class and has the same type as the member. The result is not an lvalue.

If `p_to_y` is defined as a pointer to an `int` and `y` as an `int`, the following expression assigns the address of the variable `y` to the pointer `p_to_y` :

```
p_to_y = &y;
```

> **IBM** The address operator has been extended to handle vector types, provided that vector support is enabled. The result of the address operator applied to a vector type can be stored in a pointer to a compatible vector type. The address of a vector type can be used to initialize a pointer to vector type if both sides of the initialization have compatible types. A pointer to `void` can also be initialized with the address of a vector type.

> **IBM** The address of a label can be taken using the GNU C address operator `&&`. The label can thus be used as a value.

Related reference

"Pointers" on page 79

"Indirection operator *"

"Labels as values (IBM extension)" on page 146

Indirection operator *

The * (indirection) operator determines the value referred to by the pointer-type operand. The operand cannot be a pointer to an incomplete type. If the operand points to an object, the operation yields an lvalue referring to that object. If the operand points to a function, the result is a function designator. 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.

▷ IBM The indirection operator * has been extended to handle pointer to vector
types, provided that vector support is enabled. A vector pointer should point to a
memory location that has 16-byte alignment. However, the compiler does not
enforce this constraint. Dereferencing a vector pointer maintains the vector type
and its 16-byte alignment. If a program dereferences a vector pointer that does not
contain a 16-byte aligned address, the behavior is undefined.

Related reference

"Pointers" on page 79

"Address operator &" on page 118

"Arrays" on page 83

The __alignof__ operator (IBM extension)

The __alignof__ operator is a language extension to C99 that returns the number
of bytes used in the alignment of its operand. The operand can be an expression or
a parenthesized type identifier. If the operand is an expression representing an
lvalue, the number returned by __alignof__ represents the alignment that the
lvalue is known to have. The type of the expression is determined at compile time,
but the expression itself is not evaluated. If the operand is a type, the number
represents the alignment usually required for the type on the target platform.

The __alignof__ operator may not be applied to the following:
* An lvalue representing a bit field
* A function type
* An undefined structure or class
* An incomplete type (such as void)

__alignof__ operator syntax

```
▶▶──__alignof__──┬──unary_expression──┬──────────────────────────────▶◀
                 └──(──type-id──)──────┘
```

If *type-id* is a reference or a referenced type, the result is the alignment of the
referenced type. If *type-id* is an array, the result is the alignment of the array
element type. If *type-id* is a fundamental type, the result is implementation-defined.

For example, on AIX, __alignof__(wchar_t) returns 2 for a 32-bit target, and 4 for
a 64-bit target.

The operand of __alignof__ can be a vector type, provided that vector support is
enabled. For example,

```
vector unsigned int v1 = (vector unsigned int)(10);
vector unsigned int *pv1 = &v1;
__alignof__(v1); // vector type alignment: 16.
__alignof__(&v1); // address of vector alignment: 4.
__alignof__(*pv1); // dereferenced pointer to vector alignment: 16.
__alignof__(pv1); // pointer to vector alignment: 4.
__alignof__(vector signed char); // vector type alignment: 16.
```

When `__attribute__((aligned))` is used to increase the alignment of a variable of vector type, the value returned by the `__alignof__` operator is the alignment factor specified by `__attribute__((aligned))`.

Related reference

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

►►──sizeof──┬─*expr*──────────┬──────────────────────────────►◄
            └─(─*type-name*─)─┘

The result for either kind of operand is not an lvalue, but a constant integer value. The type of the result is the unsigned integral type `size_t` defined in the header file `stddef.h`.

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 would be used by an object of that type, including any internal or trailing padding.

▶ **IBM** The operand of the `sizeof` operator can be a vector type or the result of dereferencing a pointer to vector type, provided that vector support is enabled. In these cases, the return value of `sizeof` is always 16.

```
vector bool int v1;
vector bool int *pv1 = &v1;
sizeof(v1); // vector type: 16.
sizeof(&v1); // address of vector: 4.
sizeof(*pv1); // dereferenced pointer to vector: 16.
sizeof(pv1); // pointer to vector: 4.
sizeof(vector bool int); // vector type: 16.
```

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.

The `sizeof` operator may not be applied to:

- A bit field
- A function type
- An undefined structure or class
- An incomplete type (such as `void`)

The `sizeof` operator applied to an expression yields the same result as if it had been applied to only the name of the type of the expression. At compile time, the compiler analyzes the expression to determine its type. 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`, or `int` or any enumeration type.

Related reference

"Integer constant expressions" on page 111

"Type names" on page 78

"Arrays" on page 83

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.

The `typeof` operator is extended to accept a vector type as its operand, when vector support is enabled.

typeof operator syntax

```
►►─┬─__typeof__─┬─(─┬─expr─────┬─)──────────────────────►◄
   └─typeof─────┘    └─type-name─┘
```

A `typeof` construct itself is not an expression, but the name of a type. A `typeof` construct behaves like a type name defined using `typedef`, although the syntax resembles that of `sizeof`.

The following examples illustrate its basic syntax. For an expression e:

```
int e;
__typeof__(e + 1) j;   /* the same as declaring int j;    */
e = (__typeof__(e)) f; /* the same as casting e = (int) f; */
```

Using a `typeof` construct is equivalent to declaring a `typedef` name. Given

```
int T[2];
int i[2];
```

you can write

```
__typeof__(i) a;        /* all three constructs have the same meaning */
__typeof__(int[2]) a;
__typeof__(T) a;
```

The behavior of the code is as if you had declared `int a[2];`.

For a bit field, `typeof` represents the underlying type of the bit field. For example,
`int m:2;`, the `typeof(m)` is `int`. Since the bit field property is not reserved, `n` in
`typeof(m) n;` is the same as `int n`, but not `int n:2`.

The `typeof` operator can be nested inside `sizeof` and itself. The following
declarations of `arr` as an array of pointers to `int` are equivalent:

```
int *arr[10];                    /* traditional C declaration         */
__typeof__(__typeof__ (int *)[10]) a;  /* equivalent declaration  */
```

The `typeof` operator can be useful in macro definitions where expression `e` is a
parameter. For example,

```
#define SWAP(a,b) { __typeof__(a) temp; temp = a; a = b; b = temp; }
```

Note:

1. The `typeof` and `__typeof__` keywords are supported as follows:

    - The `__typeof__` keyword is recognized under compilation with the **xlc**
      invocation command or the **-qlanglvl=extc89**, **-qlanglvl=extc99**, or
      -qlanglvl=extended options. The `typeof` keyword is only recognized under
      compilation with **-qkeyword=typeof**.

   Related reference

   "Type names" on page 78

   "typedef definitions" on page 65

   "Vector types (IBM extension)" on page 52

The __real__ and __imag__ operators (IBM extension)

XL 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

```
►►──┬──__real__──┬──(──var_identifier──)──────────────────────►◄
    └──__imag__──┘
```

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 = 2.000000 + 3.000000 * i
```

> **Related reference**
>
> "Floating-point types" on page 49
>
> Complex literals (C only)
>
> Complex floating-point types (C only)

The vec_step operator (IBM extension)

The vec_step operator takes a vector type argument and returns an integer value representing the amount by which a pointer to a vector element should be incremented to move by 16 bytes. For complete information of this operator, see the *AltiVec Technology Programming Interface Manual*, available at http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf

Binary expressions

A *binary expression* contains two operands separated by one operator. The supported binary operators are:

- "Assignment operators" on page 124
- "Multiplication operator *" on page 126
- "Division operator /" on page 126
- "Remainder operator %" on page 126
- "Addition operator +" on page 127
- "Subtraction operator -" on page 127
- "Bitwise left and right shift operators << >>" on page 127
- "Relational operators < > <= >=" on page 128
- "Equality and inequality operators == !=" on page 129
- "Bitwise AND operator &" on page 130
- "Bitwise exclusive OR operator ^" on page 131
- "Bitwise inclusive OR operator |" on page 131
- "Logical AND operator &&" on page 132
- "Logical OR operator ||" on page 132
- "Array subscripting operator [ ]" on page 133
- "Comma operator ," on page 134

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 29 on page 142.

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.

Related reference

"Integral and floating-point promotions" on page 104

"Lvalues and rvalues" on page 109

"Arithmetic conversions and promotions" on page 101

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 ="
- "Compound assignment operators" on page 125

The left operand in all assignment expressions must be a modifiable lvalue. The type of the expression is the type of the left operand. The value of the expression is the value of the left operand after the assignment has completed.

The result of an assignment expression is not an lvalue.

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*.

The left operand must be a modifiable lvalue. The type of an assignment operation is the type of the left operand.

If the left operand is not a class type or a vector type, the right operand is implicitly converted to the type of the left operand. This converted type will not be qualified by `const` or `volatile`.

If the left operand is a class type, that type must be complete. The copy assignment operator of the left operand will be called.

If the left operand is an object of reference type, the compiler will assign the value of the right operand to the object denoted by the reference.

► IBM   The assignment operator has been extended to permit operands of vector type. Both sides of an assignment expression must be of the same vector type.

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.

▶ IBM When GNU C language features have been enabled, compound expressions and conditional expressions are allowed as lvalues, provided that their operands are lvalues. The following compound assignment of the compound

expression (a, b) is legal under GNU C, provided that expression b, or more generally, the last expression in the sequence, is an lvalue:

```
(a,b) += 5   /* Under GNU C, this is equivalent to
a, (b += 5)    */
```

Related reference

"Initialization of structures and unions" on page 90

"Lvalues and rvalues" on page 109

"Pointers" on page 79

"Type qualifiers" on page 67

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.

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 80

"Pointer conversions" on page 105

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 80

"Pointer conversions" on page 105

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.

The type of the result is `int` and has the values 1 if the specified relationship is true, and 0 if false.

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. 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.

The type of the result is `int` and has the values 1 if the specified relationship is true, and 0 if false.

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 33

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 33

Logical AND operator &&

The && (logical AND) operator indicates whether both operands are true.

If both operands have nonzero values, the result has the value 1. Otherwise, the result has the value 0. The type of the result is `int`. Both operands must have a arithmetic or pointer type. The usual arithmetic conversions on each operand are performed.

Unlike the & (bitwise AND) operator, the && operator guarantees left-to-right evaluation of the operands. If the left operand evaluates to 0 (or `false`), the right operand is not evaluated.

The following examples show how the expressions that contain the logical AND operator are evaluated:

Expression	Result
1 && 0	`false` or 0
1 && 4	`true` or 1
0 && 0	`false` or 0

The following example uses the logical AND operator to avoid division by zero:

`(y != 0) && (x / y)`

The expression x / y is not evaluated when y != 0 evaluates to 0 (or `false`).

Note: The logical AND (&&) should not be confused with the bitwise AND (&) operator. For example:

   1 && 4 evaluates to 1 (or   true)

while

   1 & 4 evaluates to 0

Logical OR operator ||

The || (logical OR) operator indicates whether either operand is true.

If either of the operands has a nonzero value, the result has the value 1. Otherwise, the result has the value 0. The type of the result is `int`. Both operands must have a arithmetic or pointer type. The usual arithmetic conversions on each operand are performed.

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;
}
```

The following is the output of the above example:

```
a[0] = 10
a[1] = 20
a[2] = 30
```

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. However, using the address of a non-lvalue as an array subscript is still not allowed. The following example is valid in C99:

```
struct trio{int a[3];};
struct trio f();
foo (int index)
{
   return f().a[index];
}
```

Related reference

"Examples of declarators" on page 78

"Arrays" on page 83

"Pointers" on page 79

"Integral types" on page 47

"Lvalues and rvalues" on page 109

"Pointer arithmetic" on page 80

"Parameter declarations" on page 177

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.

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.

Related reference
"The void type" on page 51

Conditional expressions

A *conditional expression* is a compound expression that contains a condition (*operand₁*), an expression to be evaluated if the condition evaluates to true (*operand₂*), and an expression to be evaluated if the condition has the value false (*operand₃*).

The conditional expression contains one two-part operator. The ? symbol follows the condition, and the : symbol appears between the two action expressions. All expressions that occur between the ? and : are treated as one expression.

The first operand must have a scalar type. The type of the second and third operands must be one of the following:
- An arithmetic type
- A compatible pointer, structure, or union type
- void

The second and third operands can also be a pointer or a null pointer constant.

Two objects are compatible when they have the same type but not necessarily the same type qualifiers (`volatile` or `const`). Pointer objects are compatible if they have the same type or are pointers to void.

The first operand is evaluated, and its value determines whether the second or third operand is evaluated:
- If the value is true, the second operand is evaluated.
- If the value is false, the third operand is evaluated.

The result is the value of the second or third operand.

If the second and third expressions evaluate to arithmetic types, the usual arithmetic conversions are performed on the values. The types of the second and third operands determine the type of the result as shown in the following tables.

Conditional expressions have right-to-left associativity with respect to their first and third operands. The leftmost operand is evaluated first, and then only one of the remaining two operands is evaluated. The following expressions are equivalent:

```
a ? b : c ? d : e ? f : g
a ? b : (c ? d : (e ? f : g))
```

> **Related reference**
> "The void type" on page 51

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

Table 26. Types of operands and results in conditional C expressions (continued)

Type of one operand	Type of other operand	Type of result
`void`	`void`	`void`
Pointer to compatible type	Pointer to compatible type	Pointer to type with all the qualifiers specified for the type
Pointer to type	NULL pointer (the constant 0)	Pointer to type
Pointer to object or incomplete type	Pointer to `void`	Pointer to `void` with all the qualifiers specified for the type

> **IBM** In GNU C, a conditional expression is a valid lvalue, provided that its type is not `void` and both of its branches are valid lvalues. The following conditional expression (`a ? b : c`) is legal under GNU C:

```
(a ? b : c) = 5
/*  Under GNU C, equivalent to (a ? b = 5 : (c = 5))  */
```

This extension is available when compiling in one of the extended language levels.

Examples of conditional expressions

The following expression determines which variable has the greater value, `y` or `z`, and assigns the greater value to the variable `x`:

```
x = (y > z) ? y : z;
```

The following is an equivalent statement:

```
if (y > z)
   x = y;
else
   x = z;
```

The following expression calls the function `printf`, which receives the value of the variable `c`, if `c` evaluates to a digit. Otherwise, `printf` receives the character constant `'x'`.

```
printf(" c = %c\n", isdigit(c) ? c : 'x');
```

If the last operand of a conditional expression contains an assignment operator, use parentheses to ensure the expression evaluates properly. For example, the `=` operator has higher precedence than the `?:` operator in the following expression:

```
int i,j,k;
(i == 7) ? j ++ : k = j;
```

The compiler will interpret this expression as if it were parenthesized this way:

```
int i,j,k;
((i == 7) ? j ++ : k) = j;
```

That is, `k` is treated as the third operand, not the entire assignment expression `k = j`.

To assign the value of `j` to `k` when `i == 7` is false, enclose the last operand in parentheses:

```
int i,j,k;
(i == 7) ? j ++ : (k = j);
```

Cast expressions

A cast operator is used for *explicit type conversions*. It converts the value of an expression to a specified type.

Cast operator ()

Cast expression syntax

►►──(──*type*──)──*expression*──►◄

The result of this operation is not an lvalue.

The following 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 will hold an object of the size of its argument. The statement `int* myArray = (int*) malloc(10 * sizeof(int))` does the following:

- Creates a void pointer that points to memory that can hold ten integers.
- Converts that `void` pointer into an integer pointer with the use of the cast operator.
- Assigns that integer pointer to `myArray`. Because a name of an array is the same as a pointer to the initial element of the array, `myArray` is an array of ten integers stored in the memory created by the call to `malloc()`.

Cast to union type (C only) (IBM extension)

Casting to a union type is the ability to cast a union member to the same type as the union to which it belongs. Such a cast does not produce an lvalue, unlike other casts. The feature is supported as an extension to C99, implemented to facilitate porting programs developed with GNU C.

Only a type that explicitly exists as a member of a union type can be cast to that union type. The cast can use either the tag of the union type or a union type name declared in a `typedef` expression. The type specified must be a complete union type. An anonymous union type can be used in a cast to a union type, provided that it has a tag or type name. A bit field can be cast to a union type, provided that the union contains a bit field member of the same type, but not necessarily of the same length. The following shows an example of a simple cast to union:

```
#include <stdio.h>

union foo {
   char t;
   short u;
   int v;
   long w;
   long long x;
```

```
     float y;
     double z;
};

int main() {
  union foo u;
  char a = 1;
  u = (union foo)a;
  printf("u = %i\n", u.t);
}
```

The output of this example is:

```
u = 1
```

Casting to a nested union is also allowed. In the following example, the `double` type dd can be cast to the nested union u2_t.

```
int main() {
   union u_t {
      char a;
      short b;
      int c;
      union u2_t {
         double d;
      }u2;
   };
   union u_t U;
   double dd = 1.234;
   U.u2 = (union u2_t) dd;      // Valid.
   printf("U.u2 is %f\n", U.u2);
}
```

The output of this example is:

```
U.u2 is 1.234
```

A union cast is also valid as a function argument, part of a constant expression for initialization of a static or non-static data object, and in a compound literal statement. The following example shows a cast to union used as part of an expression for initializing a static object:

```
struct S
{
   int a;
};

union U {
   struct S *s;
};

struct T {
   union U u;
} t[] = {
            {(union U)&s}
  };
```

Related reference

"Structures and unions" on page 55

"The transparent_union type attribute (C only)" on page 74

"Type names" on page 78

"Lvalues and rvalues" on page 109

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.

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

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

The *type_name* can be any data type, including vectors, and 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.

Related reference

String literals

Label value expressions (IBM extension)

The label value operator && returns the address of its operand, which must be a label defined in the current function or a containing function. The value is a constant of type `void*` and should be used only in a computed goto statement. The language feature is an extension to C, implemented to facilitate porting programs developed with GNU C.

Related reference

"Labels as values (IBM extension)" on page 146

Computed goto statement

"The goto statement" on page 161

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 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 27. Precedence and associativity of postfix operators

Rank	Right associative?	Operator function	Usage
1		member selection	*object . member*
1		member selection	*pointer -> member*
1		subscripting	*pointer [expr]*
1		function call	*expr (expr_list)*
1		value construction	*type (expr_list)*
1		postfix increment	*lvalue ++*
1		postfix decrement	*lvalue --*

Table 28. Precedence and associativity of unary operators

Rank	Right associative?	Operator function	Usage
2	yes	size of object in bytes	sizeof *expr*
2	yes	size of type in bytes	sizeof (*type*)
2	yes	prefix increment	++ *lvalue*
2	yes	prefix decrement	-- *lvalue*
2	yes	bitwise negation	~ *expr*
2	yes	not	! *expr*

Table 28. Precedence and associativity of unary operators (continued)

Rank	Right associative?	Operator function	Usage
2	yes	unary minus	*- expr*
2	yes	unary plus	*+ expr*
2	yes	address of	*& lvalue*
2	yes	indirection or dereference	** expr*
2	yes	type conversion (cast)	*(type) expr*

Table 29. Precedence and associativity of binary operators

Rank	Right associative?	Operator function	Usage
3		multiplication	*expr * expr*
3		division	*expr / expr*
3		modulo (remainder)	*expr % expr*
4		binary addition	*expr + expr*
4		binary subtraction	*expr - expr*
5		bitwise shift left	*expr << expr*
5		bitwise shift right	*expr >> expr*
6		less than	*expr < expr*
6		less than or equal to	*expr <= expr*
6		greater than	*expr > expr*
6		greater than or equal to	*expr >= expr*
7		equal	*expr == expr*
7		not equal	*expr != expr*
8		bitwise AND	*expr & expr*
9		bitwise exclusive OR	*expr ^ expr*
10		bitwise inclusive OR	*expr | expr*
11		logical AND	*expr && expr*
12		logical inclusive OR	*expr || expr*
13		conditional expression	*expr ? expr : expr*
14	yes	simple assignment	*lvalue = expr*
14	yes	multiply and assign	*lvalue *= expr*
14	yes	divide and assign	*lvalue /= expr*
14	yes	modulo and assign	*lvalue %= expr*
14	yes	add and assign	*lvalue += expr*
14	yes	subtract and assign	*lvalue -= expr*
14	yes	shift left and assign	*lvalue <<= expr*
14	yes	shift right and assign	*lvalue >>= expr*
14	yes	bitwise AND and assign	*lvalue &= expr*
14	yes	bitwise exclusive OR and assign	*lvalue ^= expr*

Table 29. Precedence and associativity of binary operators (continued)

Rank	Right associative?	Operator function	Usage
14	yes	bitwise inclusive OR and assign	*lvalue |= expr*
15		comma (sequencing)	*expr , expr*

Related reference

Examples of expressions and precedence

The parentheses in the following expressions explicitly show how the compiler groups operands and operators.

```
total = (4 + (5 * 3));
total = (((8 * 5) / 10) / 3);
total = (10 + (5/3));
```

If parentheses did not appear in these expressions, the operands and operators would be grouped in the same manner as indicated by the parentheses. For example, the following expressions produce the same output.

```
total = (4+(5*3));
total = 4+5*3;
```

Because the order of grouping operands with operators that are both associative and commutative is not specified, the compiler can group the operands and operators in the expression:

```
total = price + prov_tax +
city_tax;
```

in the following ways (as indicated by parentheses):

```
total = (price + (prov_tax + city_tax));
total = ((price + prov_tax) + city_tax);
total = ((price + city_tax) + prov_tax);
```

The grouping of operands and operators does not affect the result unless one ordering causes an overflow and another does not. For example, if `price = 32767`, `prov_tax = -42`, and `city_tax = 32767`, and all three of these variables have been declared as integers, the third statement `total = ((price + city_tax) + prov_tax)` will cause an integer overflow and the rest will not.

Because intermediate values are rounded, different groupings of floating-point operators may give different results.

In certain expressions, the grouping of operands and operators can affect the result. For example, in the following expression, each function call might be modifying the same global variables.

```
a = b() + c() + d();
```

This expression can give different results depending on the order in which the functions are called.

If the expression contains operators that are both associative and commutative and the order of grouping operands with operators can affect the result of the expression, separate the expression into several expressions. For example, the following expressions could replace the previous expression if the called functions do not produce any side effects that affect the variable a.

```
a = b();
a += c();
a += d();
```

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().

Related reference

"Type names" on page 78

Chapter 8. Statements

A statement, the smallest independent computational unit, specifies an action to be performed. In most cases, statements are executed in sequence. The following is a summary of the statements available in C:

- Labeled statements
- Expression statements
- Block statements
- Selection statements
- Iteration statements
- Jump statements
- Declaration statements
- Null statement
- ▶ **IBM** Inline assembly statements (C only) (IBM extension)

  Related reference

  Chapter 4, "Data objects and declarations," on page 37

  "Function declarations" on page 170

Labeled statements

There are three kinds of labels: identifier, case, and default.

Labeled statement syntax

▶▶──*identifier*──:──*statement*──◀◀

The label consists of the *identifier* and the colon (:) character.

A label name must be unique within the function in which it appears.

Case and default label statements only appear in `switch` statements. These labels are accessible only within the closest enclosing `switch` statement.

case statement syntax

▶▶──case──*constant_expression*──:──*statement*────────────────────────◀◀

default statement syntax

▶▶──default──:──*statement*──◀◀

The following are examples of labels:
```
comment_complete : ;            /* null statement label */
test_for_null : if (NULL == pointer)
```
   Related reference

   "Function scope" on page 5

   "The goto statement" on page 161

Locally declared labels (IBM extension)

A locally declared label, or *local label*, is an identifier label that is declared at the beginning of a statement expression and for which the scope is the statement expression in which it is declared and defined. This language feature is an extension of C to facilitate handling programs developed with GNU C.

A local label can be used as the target of a `goto` statement, jumping to it from within the same block in which it was declared. This language extension is particularly useful for writing macros that contain nested loops, capitalizing on the difference between its statement scope and the function scope of an ordinary label.

Locally declared label syntax

```
►►── __label__ ──┬─ identifier ─┬── ; ──────────────────────────►◄
                 └──── , ◄──────┘
```

In a statement expression, the declaration of a local label must appear immediately after the left parenthesis and left brace, and must precede any ordinary declarations and statements. The label is defined in the usual way, with a name and a colon, within the statements of the statement expression.

Related reference

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, implemented to facilitate porting programs developed with GNU C.

In the following example, the computed goto statements use the values of `label1` and `label2` to jump to those spots in the function.

```
int main()
{
   void * ptr1, *ptr2;
   ...
   label1: ...
   ...
   label2: ...
   ...
   ptr1 = &&label1;
   ptr2 = &&label2;
   if (...) {
      goto *ptr1;
   } else {
      goto *ptr2;
   }
   ...
}
```

Related reference

Expression statements

An *expression statement* contains an expression. The expression can be null.

Expression statement syntax

```
>>────────────────;──────────────────────────────────><
      └─expression─┘
```

An expression statement evaluates *expression*, then discards the value of the expression. An expression statement without an expression is a null statement.

The following are 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 7, "Expressions and operators," on page 109

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

```
>>──{─┬─────────────────────────────┬─┬─────────────┬─}──><
      │  ┌──────────────────────────┐│ │ ┌──────────┐│
      ├─type_definition──────────────┤ └─statement──┘
      ├─file_scope_data_declaration──┤
      └─block_scope_data_declaration─┘
```

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**
>
> "Block scope" on page 4
>
> "The auto storage class specifier" on page 41

Example of blocks

The following program shows how the values of data objects change in nested blocks:

```
/**
** This example shows how data objects change in nested blocks.
**/
 #include <stdio.h>
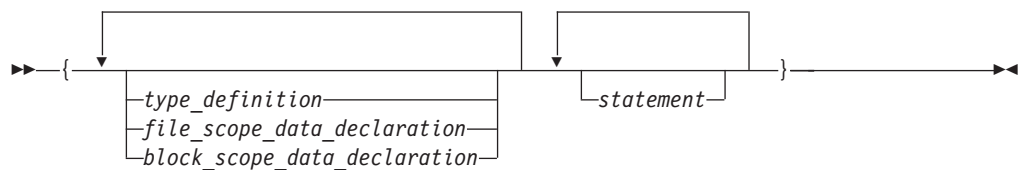
 int main(void)
 {
   int x = 1;                         /* Initialize x to 1  */
   int y = 3;

   if (y > 0)
   {
      int x = 2;                      /* Initialize x to 2  */
      printf("second x = %4d\n", x);
   }
   printf("first  x = %4d\n", x);

   return(0);
 }
```

The program produces the following output:

```
second x =    2
first  x =    1
```

Two variables named x are defined in main. The first definition of x retains storage while main is running. However, because the second definition of x occurs within a nested block, printf("second x = %4d\n", x); recognizes x as the variable defined on the previous line. Because printf("first x = %4d\n", x); is not part of the nested block, x is recognized as the first definition of x.

Statement expressions (IBM extension)

A compound statement is a sequence of statements enclosed by braces. In GNU C, a compound statement inside parentheses may appear as an expression in what is called a *statement expression*.

Statement expression syntax

The value of a statement expression is the value of the last simple expression to appear in the entire construct. If the last statement is not an expression, then the construct is of type void and has no value.

The statement expression can be combined with the typeof operator to create complex function-like macros in which each operand is evaluated only once. For example:

```
#define SWAP(a,b) ( {__typeof__(a) temp; temp=a; a=b; b=temp;} )
```

Related reference

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.

In C, an `if` statement lets you conditionally process a statement when the specified test expression evaluates to a nonzero value. The test expression must be of arithmetic or pointer type.

You can optionally specify an `else` clause on the `if` statement. If the test expression evaluates to a zero value and an `else` clause exists, the statement associated with the `else` clause runs. If the test expression evaluates to 1, the statement following the expression runs and the `else` clause is ignored.

if statement syntax

```
►►──if──(──expression──)──statement──┬──────────────────┬──►◄
                                     └─else──statement──┘
```

When `if` statements are nested and `else` clauses are present, a given `else` is associated with the closest preceding `if` statement within the same block.

A single statement following any selection statements (`if`, `switch`) is treated as a compound statement containing the original statement. As a result any variables declared on that statement will be out of scope after the `if` statement. For example:

```
if (x)
int i;
```

is equivalent to:

```
if (x)
{  int i; }
```

Variable `i` is visible only within the `if` statement. The same rule applies to the `else` part of the `if` statement.

Examples of if statements

The following example causes `grade` to receive the value A if the value of `score` is greater than or equal to 90.

```
if (score >= 90)
   grade = 'A';
```

The following example displays Number is positive if the value of `number` is greater than or equal to 0. If the value of `number` is less than 0, it displays Number is negative.

```
if (number >= 0)
   printf("Number is positive\n");
else
   printf("Number is negative\n");
```

The following example shows a nested `if` statement:

```
if (paygrade == 7)
   if (level >= 0 && level <= 8)
      salary *= 1.05;
   else
      salary *= 1.04;
else
   salary *= 1.06;
cout << "salary is " << salary << endl;
```

The following example shows a nested `if` statement that does not have an `else` clause. Because an `else` clause always associates with the closest `if` statement, braces might be needed to force a particular `else` clause to associate with the correct `if` statement. In this example, omitting the braces would cause the `else` clause to associate with the nested `if` statement.

```
if (kegs > 0) {
   if (furlongs > kegs)
      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

Chapter 6, "Type conversions," on page 101

"Boolean types" on page 48

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

```
▶▶──switch──(──expression──)──switch_body──────────────────────────────────────▶◀
```

The *switch body* is enclosed in braces and can contain definitions, declarations, *case clauses*, and a *default clause*. Each case clause and default clause can contain statements.

```
▶▶──{─┬────────────────────────────┬──┬─────────────────┬────────────────────────▶
      │ ┌────────────────────────┐ │  │ ┌─────────────┐ │
      └─┼──type_definition────────┼─┘  └─┴──case_clause─┴─┘
        ├──file_scope_data_declaration──┤
        └──block_scope_data_declaration─┘
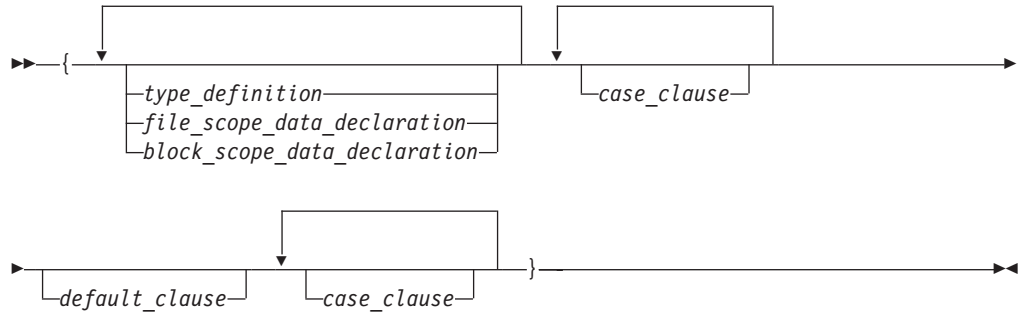
────┬───────────────────┬──┬─────────────────┬──}─────────────────────────────────▶◀
    └──default_clause────┘  │ ┌─────────────┐ │
                            └─┴──case_clause─┴─┘
```

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

```
                 ┌───────────┐
▶▶──case_label───┴──statement─┴──────────────────────────────────────────────▶◀
```

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

```
    ┌──────────────────────────────────────┐
▶▶──┴──case──integral_constant_expression──:─┴────────────────────────────────▶◀
```

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

```
▶▶──┬───────────────┬──default──:──┬───────────────┬──┬───────────┬──────────────▶◀
    └──case_label────┘              └──case_label────┘  │ ┌───────┐ │
                                                        └─┴─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.

Compiling with option **-qinfo=gen** finds case labels that fall through when they should not.

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.

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.

Examples of switch statements

The following switch statement contains several case clauses and one default clause. Each clause contains a function call and a break statement. The break statements prevent control from passing down through each statement in the switch body.

If the switch expression evaluated to '/', the switch statement would call the function divide. Control would then pass to the statement following the switch body.

```
char key;

printf("Enter an arithmetic operator\n");
scanf("%c",&key);

switch (key)
{
   case '+':
      add();
      break;
```

```
      case '-':
         subtract();
         break;

      case '*':
         multiply();
         break;

      case '/':
         divide();
         break;

      default:
         printf("invalid key\n");
         break;
}
```

If the switch expression matches a case expression, the statements following the case expression are processed until a break statement is encountered or the end of the switch body is reached. In the following example, break statements are not present. If the value of text[i] is equal to 'A', all three counters are incremented. If the value of text[i] is equal to 'a', lettera and total are increased. Only total is increased if text[i] is not equal to 'A' or 'a'.

```
char text[100];
int capa, lettera, total;

// ...

for (i=0; i<sizeof(text); i++) {

    switch (text[i])
    {
       case 'A':
         capa++;
       case 'a':
         lettera++;
       default:
         total++;
    }
}
```

The following switch statement performs the same statements for more than one case label:

```
/**
 ** This example contains a switch statement that performs
 ** the same statement for more than one case label.
 **/

#include <stdio.h>

int main(void)
{
  int month;

  /* Read in a month value */
  printf("Enter month: ");
  scanf("%d", &month);

  /* Tell what season it falls into */
  switch (month)
  {
     case 12:
     case 1:
```

```
        case 2:
           printf("month %d is a winter month\n", month);
           break;

        case 3:
        case 4:
        case 5:
           printf("month %d is a spring month\n", month);
           break;

        case 6:
        case 7:
        case 8:
           printf("month %d is a summer month\n", month);
           break;

        case 9:
        case 10:
        case 11:
           printf("month %d is a fall month\n", month);
           break;

        case 66:
        case 99:
        default:
           printf("month %d is not a valid month\n", month);
   }

  return(0);
}
```

If the expression month has the value 3, control passes to the statement:

```
printf("month %d is a spring month\n", month);
```

The break statement passes control to the statement following the switch body.

> **Related reference**
>
> Chapter 6, "Type conversions," on page 101
>
> "Labeled statements" on page 145
>
> See -qinfo=gen in the XL C Compiler Reference
>
> Case and default labels
>
> "The break statement" on page 158

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 48

The while statement

A *while statement* repeatedly runs the body of a loop until the controlling expression evaluates to 0.

while statement syntax

▶▶──while──(──*expression*──)──*statement*──────────────────────────────────────▶◀

The *expression* must be of arithmetic or pointer type.

The expression is evaluated to determine whether or not to process the body of the loop. If the expression evaluates to 0, the body of the loop never runs. If the expression does not evaluate to 0, 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 0.

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.

```
/**
 ** 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 0. Because of the order of processing, the statement is run at least once.

do statement syntax

▶▶──do──*statement*──while──(──*expression*──)──;──────────────────────────────▶◀

The *expression* must be of arithmetic or pointer type.

The body of the loop is run before the controlling `while` clause is evaluated. Further processing of the `do` statement depends on the value of the `while` clause. If the `while` clause does not evaluate to 0, the statement runs again. When the `while` clause evaluates to 0, 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 0.

The following example keeps incrementing i while i is less than 5:

```
#include <stdio.h>

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

The following is the output of the above example:

```
Value of i: 1
Value of i: 2
Value of i: 3
Value of i: 4
Value of i: 5
```

The for statement

A *for statement* lets you do the following:
- Evaluate an expression before the first iteration of the statement (*initialization*)
- Specify an expression to determine whether or not the statement should be processed (the *condition*)
- Evaluate an expression after each iteration of the statement (often used to increment for each iteration)
- Repeatedly process the statement if the controlling part does not evaluate to 0.

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*. *expression2* must be of arithmetic or pointer type.

If it evaluates to 0, the statement is not processed and control moves to the next statement following the for statement. If *expression2* does not evaluate to 0, the statement is processed. If you omit *expression2*, it is as if the expression had been replaced by 1, 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 0. 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
- The goto statement
- ▶ IBM   Computed goto statement (IBM extension)

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

►►—break—;———————————————————————————————►◄

In an iterative statement, the break statement ends the loop and moves control to the next statement outside the loop. Within nested statements, the break statement ends only the smallest enclosing do, for, switch, or while statement.

In a switch statement, the break passes control out of the switch body to the next statement outside the switch statement.

> **Related reference**
> "The switch statement" on page 150

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:

►►—continue—;———————————————————————————————►◄

A continue statement can only appear within the body of an iterative statement, such as do, for, or while.

The continue statement ends the processing of the action part of an iterative statement and moves control to the loop continuation portion of the statement. For

example, if the iterative statement is a for statement, control moves to the third expression in the condition part of the statement, then to the second expression (the test) in the condition part of the statement.

Within nested statements, the continue statement ends only the current iteration of the do, for, or while statement immediately enclosing it.

Examples of continue statements

The following example shows a continue statement in a for statement. The continue statement causes processing to skip over those elements of the array rates that have values less than or equal to 1.

```
/**
 ** This example shows a continue statement in a for statement.
 **/

#include <stdio.h>
#define  SIZE  5

int main(void)
{
   int i;
   static float rates[SIZE] = { 1.45, 0.05, 1.88, 2.00, 0.75 };

   printf("Rates over 1.00\n");
   for (i = 0; i < SIZE; i++)
   {
      if (rates[i] <= 1.00)  /*  skip rates <= 1.00  */
         continue;
      printf("rate = %.2f\n", rates[i]);
   }

   return(0);
}
```

The program produces the following output:

```
Rates over 1.00
rate = 1.45
rate = 1.88
rate = 2.00
```

The following example shows a continue statement in a nested loop. When the inner loop encounters a number in the array strings, that iteration of the loop ends. Processing continues with the third expression of the inner loop. The inner loop ends when the '\0' escape sequence is encountered.

```
/**
 ** This program counts the characters in strings that are part
 ** of an array of pointers to characters.  The count excludes
 ** the digits 0 through 9.
 **/

#include <stdio.h>
#define  SIZE  3

int main(void)
{
   static char *strings[SIZE] = { "ab", "c5d", "e5" };
   int i;
   int letter_count = 0;
   char *pointer;
   for (i = 0; i < SIZE; i++)              /* for each string   */
                                           /* for each each character */
```

```
        for (pointer = strings[i]; *pointer != '\0';
        ++pointer)
        {                                       /* if a number      */
           if (*pointer >= '0' && *pointer <= '9')
              continue;
           letter_count++;
        }
     printf("letter count = %d\n", letter_count);

     return(0);
}
```

The program produces the following output:

```
letter count = 5
```

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.

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.

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**
>
> Chapter 6, "Type conversions," on page 101
>
> "Function return type specifiers" on page 176
>
> "Function return values" on page 176

The goto statement

A *goto statement* causes your program to unconditionally transfer control to the statement associated with the label specified on the `goto` statement.

goto statement syntax

►►—goto—*label_identifier*—;————————————————————►◄

Because the `goto` statement can interfere with the normal sequence of processing, it makes a program more difficult to read and maintain. Often, a `break` statement, a `continue` statement, or a function call can eliminate the need for a `goto` statement.

If an active block is exited using a `goto` statement, any local variables are destroyed when control is transferred from that block.

You cannot use a `goto` statement to jump over initializations.

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.

The following example shows a `goto` statement that is used to jump out of a nested loop. This function could be written without using a `goto` statement.

```
/**
 ** This example shows a goto statement that is used to
 ** jump out of a nested loop.
 **/

#include <stdio.h>
void display(int matrix[3][3]);

int main(void)
{
   int matrix[3][3]=    {1,2,3,4,5,2,8,9,10};
   display(matrix);
   return(0);
}

void display(int matrix[3][3])
{
   int i, j;

   for (i = 0; i < 3; i++)
      for (j = 0; j < 3; j++)
      {
         if ( (matrix[i][j] < 1) || (matrix[i][j] > 6) )
            goto out_of_bounds;
         printf("matrix[%d][%d] = %d\n", i, j, matrix[i][j]);
      }
   return;
   out_of_bounds: printf("number must be 1 through 6\n");
}
```

Computed goto 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, implemented to facilitate porting programs developed with GNU C.

Computed goto statement syntax

►►──goto──*expression*──;──►◄

The *expression* is an expression of type `void*`.

> **Related reference**
> "The auto storage class specifier" on page 41
> "Labeled statements" on page 145
> "Labels as values (IBM extension)" on page 146
> "Label value expressions (IBM extension)" on page 140

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)

Under extended language levels, the compiler provides full support for embedded assembly code fragments among C source statements. This extension has been implemented for use in general system programming code, and in the operating system kernel and device drivers, which were originally developed with GNU C.

The keyword `asm` stands for assembly code. When strict language levels are used in compilation, the C compiler recognizes and ignores the keyword `asm` in a declaration.

The syntax is as follows:

asm statement syntax — statement in local scope

input:

output:

asm statement syntax — statement in global scope

```
>>──┬─asm────┬──(──code_format_string──)─────────────────────><
    ├──asm───┤
    └─ __asm__ ─┘
```

The qualifier `volatile` instructs the compiler that the assembler instructions may update memory not listed in *output*, *input*, or *clobbers*.

The *code_format_string* is the source text of the `asm` instructions and is a string literal similar to a `printf` format specifier. Depending on the operands taken by the assembly instruction, the string contains zero or more modifiers, each of which corresponds to an input or output operand, separated by commas.

The `%` modifier can take either of the following forms:
- `%integer`, where *integer* refers to the sequential number of the input or output operand.
- `%[symbolic_name]`, where the *symbolic_name* is referenced in the operand list. The symbolic operand names have no relation to any C identifiers. Any name can be used, even those of existing C symbols. However, no two operands in the same assembly statement can use the same symbolic name.

The *input* consists of zero, one or more input operands, separated by commas. Each operand consists of a `constraint(C_expression)` pair.

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), and, optionally, by an additional % or & modifier.

The *modifier* is one of the following:

=   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.

%   Declares the instruction to be commutative for this operand and the following operand. This means that the order of this operand and the next may be swapped when generating the instruction. This modifier can be used on an input or output operand, but cannot be specified on the last operand.

The *constraint* is a string literal that describes the kind of operand that is permitted, one character per constraint. The following constraints are supported:

b   Use a general register other than zero.

c   Use the CTR register.

f   Use a floating-point register.

g   Use a general register, memory, or immediate operand.

h   Use the CTR or LINK register.

i	Use an immediate integer or string literal operand.
l	Use the CTR register.
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.
v	Use a vector register.

0, 1, 2, ...
     A matching constraint. Allocate the same register in output as in the corresponding input.

I, J, K, L, M, N, O, P
     Constant values. Fold the expression in the operand and substitute the value into the % specifier. These constraints specify a maximum value for the operand, as follows:

- I — signed 16-bit
- J — unsigned 16-bit shifted left 16 bits
- K — unsigned 16-bit constant
- L — signed 16-bit shifted left 16 bits
- M — unsigned constant greater than 31
- N — unsigned constant that is an exact power of 2
- O — zero
- P — signed whose negation is a signed 16-bit constant

The *C_expression* is a 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 should honor the ANSI aliasing rule (see "Type-based aliasing" on page 81 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 **-qalias=ansi** option during compilation.

clobbers is a comma-separated list of register names enclosed in double quotes. These are registers that can be updated by the `asm` instruction. The following are valid register names:

r0 to r31
     General purpose registers

f0 to f31
     Floating-point registers

lr	Link register
ctr	Loop count, decrement and branching register
fpscr	Floating-point status and control register
xer	Fixed-point exception register

cr0 to cr7
> Condition registers

v0 to v31
> Vector registers (on selected processors only)

Related reference

"Identifiers" on page 12

"The register storage class specifier" on page 44

Examples of inline assembly statements

In the following example:

```
int a ;
int b = 100 ;
int c = 200 ;
 asm("add %0, %1, %2" : "=r"(a) : "r"(b) , "r"(c));
```

add is the op code of the instruction, understood by the assembler. %0, %1 and %2 are the operands, which are to be substituted by the C expressions in the output/input operand fields. The output operand uses the = constraint to indicate that a modifiable operand is required; and the r constraint to indicate that a general purpose register is required. Likewise, the r in the input operands indicates that general purpose registers are required. Within these restrictions, the compiler is free to choose any registers to substitute for %0, %1, and %2.

In the following example, the % constraint modifier tells the compiler that operands a and b can be switched if the compiler thinks it can generate better code in doing so.

```
asm("add %0, %1, %2" : "=r"(c) : "%r"(a), "r"(b));
```

The following example illustrates the use of the symbolic names for input and output operands :

```
int a ;
int b = 1, c = 2, d = 3 ;
__asm("addc %[result],%[first],%[second]" : [result]"=r"(a) : [first]"r"(b), [second]"r"(d));
```

Restrictions on inline assembly statements

The following are restrictions 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.
* Referencing an external symbol directly, without going through the operand list, is not supported.
* Assembler instructions requiring a pair of registers are not specifiable by any constraints, and are therefore not supported. For example, you can not use the %f constraint for a _Decimal128 operand.

Related reference

Assembly labels (IBM extension)

Variables in specified registers (IBM extension)

See -qasm in the XL C Compiler Reference

Chapter 9. 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 programs may not produce consistent outputs for all inputs, may not produce output at all, or may have side effects. Functions can be understood as user-defined operations, in which the parameters of the parameter list, if any, are the operands.

Information on functions include:
- Function declarations and definitions
- Function storage class specifiers
- Function specifiers
- Function return type specifiers
- Function declarators
- Function attributes (IBM extension)
- The main() function
- Function calls
- Pointers to functions
- Nested functions (IBM extension)

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.

Related reference

"Identifiers" on page 12

"Overview of data declarations and definitions" on page 39

"The void type" on page 51

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.

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.

The elements of a declaration for a function are as follows:
- "Function storage class specifiers" on page 173, which specify linkage
- "Function return type specifiers" on page 176, which specify the data type of a value to be returned
- "Function specifiers" on page 173, which specify additional properties for functions
- "Function declarators" on page 177, which include function identifiers as well as lists of parameters

All function declarations have the form:

Function declaration syntax

```
►►──────────────────────────────────────────┬─return_type_specifier───────►
     └─storage_class_specifier──function_specifier─┘
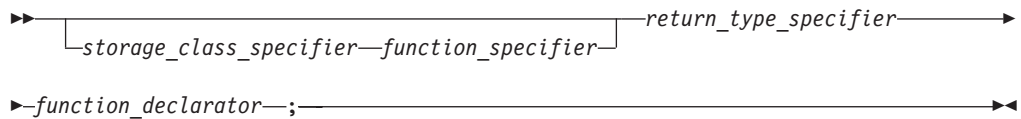
►──function_declarator──;─────────────────────────────────────────────────►◄
```

> **IBM**   In addition, for compatibility with GNU C, XL C allows you to use *attributes* to modify the properties of functions. They are described in "Function attributes (IBM extension)" on page 179.

> **Related reference**
> "Function prototype scope" on page 5
> Chapter 8, "Statements," on page 145

Function definitions

The elements of a function definition are as follows:
- "Function storage class specifiers" on page 173, which specify linkage
- "Function return type specifiers" on page 176, which specify the data type of a value to be returned
- "Function specifiers" on page 173, which specify additional properties for functions
- "Function declarators" on page 177, 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

> **IBM** In addition, for compatibility with GNU C, XL C allows you to use *attributes* to modify the properties of functions. They are described in "Function attributes (IBM extension)" on page 179.

Function definitions take the following form:

Function definition syntax (C only)

```
>>─┬──────────────────────┬──┬───────────────────┬──────────────────>
   └─storage_class_specifier─┘  └─function_specifier─┘
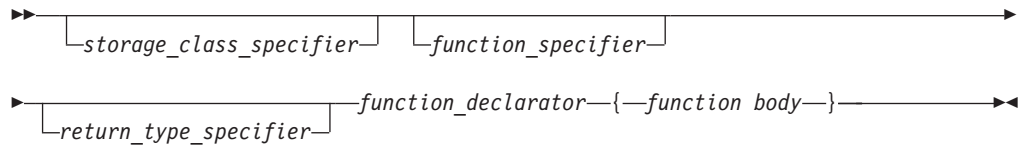
>──┬──────────────────────┬──function_declarator─{─function body─}──><
   └─return_type_specifier─┘
```

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.

```
int extern f2(const int, ...);
```

Function f6 is a const class member function of class X, takes no arguments, and has a return type of int:

```
class X
{
public:
    int f6() const;
};
```

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 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:

```
double maximum(int n, int m, double a[n][m]);
double maximum(int n, int m, double a[*][*]);
double maximum(int n, int m, double a[ ][*]);
double maximum(int n, int m, double a[ ][m]);
```

Related reference

"Overview of data objects" on page 37

Compatible and composite types

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. An exception is inline functions, which are treated by default as having internal linkage; see "Linkage of inline functions" on page 174 for more information.

A storage class specifier may be used in both function declarations and definitions. The only storage class options for functions are:

- static
- extern

Related reference

"Program linkage" on page 7

"Storage class specifiers" on page 41

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.

Related reference

"The static storage class specifier" on page 42

"Internal linkage" on page 7

"The inline function specifier"

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.

Related reference

"The extern storage class specifier" on page 43

"External linkage" on page 8

"The inline function specifier"

"Pointers to functions" on page 187

Function specifiers

The only available function specifier for functions declarations and definitions is inline, described below.

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.

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.

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: The `inline` specifier is represented by the following keywords:

- The `inline` keyword is recognized under compilation with **xlc** or **c99**, or with the **-qlanglvl=stdc99** or **-qlanglvl=extc99** options or **-qkeyword=inline**. The `__inline__` keyword is recognized at all language levels; however, see "Linkage of inline functions" below for the semantics of this keyword.

Linkage of inline functions

▶ ⬛ 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.

▶ **IBM** If you specify the __inline__ keyword, with the trailing underscores, the compiler uses the GNU C semantics for inline functions. In contrast to the C99 semantics, a function defined as __inline__ provides an external definition only; a function defined as static __inline__ provides an inline definition with internal linkage (as in C99); and a function defined as extern __inline__, when compiled with optimization enabled, allows the co-existence of an inline and external definition of the same function. For more information on the GNU C implementation of inline functions, see the GCC information, available at http://gcc.gnu.org/onlinedocs/. **IBM** ◀

Related reference

"The always_inline function attribute (IBM extension)" on page 181

"The noinline function attribute" on page 183

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*.

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.

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();
```

The user-defined type may also be defined within the function declaration.

```
enum count{one, two, three} counter();   // legal
```

Related reference

"Type specifiers" on page 47

"The return statement" on page 160

Function return values

If a function is defined as having a return type of void, it should not return a value.

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.

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 to an automatic variable should not be returned.

Related reference

"The return statement" on page 160

"The auto storage class specifier" on page 41

Function declarators

Function declarators consist of the following elements:

- An *identifier*, or name
- "Parameter declarations," which specify the parameters that can be passed to the function in a function call

Function declarator syntax (C only)

```
►►──identifier──(─┬───────────────────────┬─)────────────────────►◄
                  └─parameter_declaration─┘
```

Related reference

"Examples of declarators" on page 78

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.

Function parameter declaration syntax

```
              ┌─,────────┐
              │          │
►►──(─────────▼──────────┬───┬────────┬─)───────────────────────►◄
              └─parameter─┘   └─,──...─┘
```

parameter

```
►►──┬──────────┬──type_specifier──┬────────────┬────────────────►◄
    └─register─┘                  └─declarator─┘
```

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 should define the function with the keyword void.

An ellipsis at the end of the parameter specifications is used to specify that a function has a variable number of parameters. The number of parameters is equal to, or greater than, the number of parameter specifications.

```
    int f(int, ...);
```

At least one parameter declaration, as well as a comma before the ellipsis, are both required in C.

Parameter types

In a function *declaration*, or prototype, the type of each parameter must be specified. In the function *definition*, if the type of a parameter is not specified, it is assumed to be int.

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);
```

The user-defined type can also be defined within the parameter declaration.

```
void print(struct X { int i; } x);   // legal
```

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);
```

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.

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 42

"Arrays" on page 83

"Array subscripting operator [ ]" on page 133

"The void type" on page 51

"Type specifiers" on page 47

"Type qualifiers" on page 67

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.

IBM C implements a subset of the GNU C function attributes. If a particular function attribute is not implemented, its specification is accepted and the semantics are ignored. These language features are collectively available when compiling in any of the extended language levels.

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 declarator*─── __attribute__ ─────────────────────────────────►

►─((─┬──┬─*attribute_name*──────┬─┬─))─;─────────────────────────────────────►◄
     │ └─ __*attribute_name*─ __ ─┘ │
     └──────,──────────────────────┘

Function attribute syntax: function definition (C only)

►►─── __attribute__ ─((─┬──┬─*attribute name*──────┬─┬─))─ *function_declarator*───►
                        │ └─ __*attribute_name*─ __ ─┘ │
                        └──────,──────────────────────┘

►–{—*function body*—}————————————————————————————◄◄

Function attribute syntax: function definition (C++ only)

►►—*function declarator*—__attribute__————————————————————►

```
               ,
       ┌───────◄──────┐
►–((—▼—┬—attribute_name———┬——))—;——————————————————◄◄
       └—__—attribute_name—__—┘
```

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++ In C++, the attribute specification must also follow any exception declaration that may be present for the function.

► 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) { }
```

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

The following function attributes are supported:
- The alias function attribute (IBM extension)
- The always_inline function attribute (IBM extension)
- The format function attribute (IBM extension)
- The format_arg function attribute (IBM extension)
- The noinline function attribute (IBM extension)
- The noreturn function attribute (IBM extension)
- The pure function attribute (IBM extension)
- The weak function attribute (IBM extension)

  Related reference

  "Type attributes (IBM extension)" on page 72

  "Variable attributes (IBM extension)" on page 96

The alias function attribute

The `alias` function attribute causes the function declaration to appear in the object file as an alias for another symbol. This language feature provides a technique for coping with duplicate or cumbersome names.

alias function attribute syntax

```
►►──__attribute__──((──┬─alias───┬──(──"original_function_name"──)──))──────────►◄
                        └─__alias__─┘
```

The aliased function can be defined after the specification of its alias with this function attribute. C also allows an alias specification in the absence of a definition of the aliased function in the same compilation unit.

The following declares bar to be an alias for __foo:

```
/* C only */
void __foo(){   /*  function body  */   }
void bar() __attribute__((alias("__foo")));
```

The compiler does not check for consistency between the declaration of bar and definition of __foo. Such consistency remains the responsibility of the programmer.

Related reference

"Identifiers" on page 12

"The weak function attribute" on page 184

See #pragma weak in the XL C Compiler Reference

"The weak variable attribute" on page 99

The always_inline function attribute (IBM extension)

The always_inline function attribute instructs the compiler to inline an inline function, regardless of whether optimization was specified at compile time. However, the attribute has no effect if the program is compiled at no-opt levels. Specifying this attribute for a function without an inline specification also has no effect. The attribute takes precedence over inlining compiler options.

always_inline function attribute syntax

```
►►──__attribute__──((──┬─always_inline───┬──))──────────────────────────────────►◄
                        └─__always_inline__─┘
```

Related reference

"The inline function specifier" on page 173

"The noinline function attribute" on page 183

The const function attribute

The const function attribute allows you to tell the compiler that the function can safely be called fewer times than indicated in the source code. The language feature provides you with an explicit way to help the compiler optimize code by indicating that the function does not examine any values except its arguments and has no effects except for its return value.

const function attribute syntax

```
►►──__attribute__──((──┬─const───┬──))──────────────────────────────────────────►◄
                        └─__const__─┘
```

The following kinds of functions should not be declared const:
- A function with pointer arguments which examines the data pointed to.
- A function that calls a non-const function.

  Related reference

  See #pragma qisolated call in the XL C Compiler Reference

The format function attribute

The format function attribute provides a way to identify user-defined functions that take format strings as arguments so that calls to these functions will be type-checked against a format string, similar to the way the compiler checks calls to the functions printf, scanf, strftime, and strfmon for errors.

format function attribute syntax

where

string_index
> Is a constant integral expression that specifies which argument in the declaration of the user function is the format string argument.

first_to_check
> Is a constant integral expression that specifies the first argument to check against the format string. If there are no arguments to check against the format string (that is, diagnostics should only be performed on the format string syntax and semantics), *first_to_check* should have a value of 0. For strftime-style formats, *first_to_check* is required to be 0.

It is possible to specify multiple format attributes on the same function, in which case, all apply.

```
void my_fn(const char* a, const char* b, ...)
        __attribute__((__format__(__printf__,1,0), __format__(__scanf__,2,3)));
```

It is also possible to diagnose the same string for different format styles. All styles are diagnosed.

```
void my_fn(const char* a, const char* b, ...)
        __attribute__((__format__(__printf__,2,3),
                       __format__(__strftime__,2,0),
                       __format__(__scanf__,2,3)));
```

The format_arg function attribute

The format_arg function attribute provides a way to identify user-defined functions that modify format strings. Once the function is identified, calls to functions like printf, scanf, strftime, or strfmon, whose operands are a call to the user-defined function can be checked for errors.

format_arg function attribute syntax

```
►►──__attribute__──((──┬─format_arg───┬──(──string_index──)──))──────────────►◄
                       └─__format_arg__─┘
```

where *string_index* is a constant integral expression that specifies which argument is the format string argument, starting from 1.

It is possible to specify multiple `format_arg` attributes on the same function, in which case, all apply.

```
extern char* my_dgettext(const char* my_format, const char* my_format2)
        __attribute__((__format_arg__(1))) __attribute__((__format_arg__(2)));

printf(my_dgettext("%","%"));
//printf-style format diagnostics are performed on both "%" strings
```

The noinline function attribute

The `noinline` function attribute prevents the function to which it is applied from being inlined, regardless of whether the function is declared inline or non-inline. The attribute takes precedence over inlining compiler options, the `inline` keyword, and the `always_inline` function attribute.

noinline function attribute syntax

```
►►──__attribute__──((──┬─noinline───┬──))──────────────────────────────►◄
                       └─__noinline__─┘
```

Other than preventing inlining, the attribute does not remove the semantics of inline functions.

Related reference

"The inline function specifier" on page 173

"The always_inline function attribute (IBM extension)" on page 181

The noreturn function attribute

The `noreturn` function attribute allows you to indicate to the compiler that the function is not intended to return. The language feature provides the programmer with another explicit way to help the compiler optimize code and to reduce false warnings for uninitialized variables.

The return type of the function should be `void`.

noreturn function attribute syntax

```
►►──__attribute__──((──┬─noreturn───┬──))──────────────────────────────►◄
                       └─__noreturn__─┘
```

Registers saved by the calling function may not necessarily be restored before calling the nonreturning function.

Related reference

See #pragma_leaves in the XL C Compiler Reference

The pure function attribute

The `pure` function attribute allows you to declare a function that can be called fewer times than what is literally in the source code. Declaring a function with the attribute `pure` indicates that the function has no effect except a return value that depends only on the parameters, global variables, or both.

pure function attribute syntax

```
►►──__attribute__──((──┬──pure────┬──))──────────────────────────────────────►◄
                       └──__pure__─┘
```

Related reference

See #pragma isolated call in the XL C Compiler Reference

The weak function attribute

The `weak` function attribute causes the symbol resulting from the function declaration to appear in the object file as a weak symbol, rather than a global one. The language feature provides the programmer writing library functions with a way to allow function definitions in user code to override the library function declaration without causing duplicate name errors.

weak function attribute syntax

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

Related reference

"The weak variable attribute" on page 99

See #pragma weak in the XL C Compiler Reference
"The alias function attribute" on page 180

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`.
- `main` cannot be called from within a program.
- The address of `main` cannot be taken.
- The `main` function cannot be overloaded.

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:

```c
#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 with the following:

```
backward string1 string2
```

gives the following output:

```
string2 string1
```

The arguments argc and argv would contain the following values:

Object	Value
argc	3
argv[0]	pointer to string "backward"
argv[1]	pointer to string "string1"
argv[2]	pointer to string "string2"
argv[3]	NULL

Related reference

"The extern storage class specifier" on page 43

"The inline function specifier" on page 173

"The static storage class specifier" on page 42

"Function calls"

Function calls

Once a function has been 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, and the process just described is called *passing arguments* to the function.

Passing arguments can be done in two ways:

- "Pass by value," which copies the *value* of an argument to the corresponding parameter in the called function
- "Pass by reference," which passes the *address* of an argument to the corresponding parameter in the called function

  Related reference

  "Function argument conversions" on page 107

  "Function call expressions" on page 113

  "The main() function" on page 184

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 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 reference

Passing by *by reference* refers to a method of passing the address of an argument in the calling function to a corresponding parameter in the called function. In C, the corresponding parameter in the called function must be declared as a pointer type.

In this way, the value of the argument in the calling function can be modified by the called function.

The following example shows how arguments are passed by reference. Note that pointer parameters are initialized with pointer values when the function is called.

▶ C

```
#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;
}
```

C ◀

When the function swapnum() is called, the actual values of the variables a and b are exchanged because they are passed by reference. The output is:

```
A is 20 and B is 10
```

Pointers to functions

A pointer to a function points to the address of the executable code of the function. You can use pointers to call functions and to pass functions as arguments to other functions. You cannot perform pointer arithmetic on pointers to functions.

The type of a pointer to a function is based on both the return type and parameter types of the function.

A declaration of a pointer to a function must have the pointer name in parentheses. The function call operator () has a higher precedence than the dereference operator *. Without them, the compiler interprets the statement as a function that returns a pointer to a specified return type. For example:

```
int *f(int a);        /* function f returning an int*                    */
int (*g)(int a);      /* pointer g to a function returning an int        */
char (*h)(int, int)   /* h is a function
                         that takes two integer parameters and returns char */
```

In the first declaration, f is interpreted as a function that takes an int as argument, and returns a pointer to an int. In the second declaration, g is interpreted as a pointer to a function that takes an int argument and that returns an int.

Related reference

"Pointers" on page 79

"Pointer conversions" on page 105

"The extern storage class specifier" on page 173

Nested functions (IBM extension)

▶ C ◾ A nested function is a function defined inside the definition of another function. It can be defined wherever a variable declaration is permitted, which allows nested functions within nested functions. Within the containing function, the nested function can be declared prior to being defined by using the `auto` keyword. Otherwise, a nested function has internal linkage. The language feature is an extension to C89 and C99, implemented to facilitate porting programs developed with GNU C.

A nested function can access all identifiers of the containing function that precede its definition.

A nested function must not be called after the containing function exits.

A nested function cannot use a `goto` statement to jump to a label in the containing function, or to a local label declared with the `__label__` keyword inherited from the containing function. ◾ C ◀

Related reference

"Locally declared labels (IBM extension)" on page 146

Chapter 10. Preprocessor directives

The preprocessor is a program that is invoked by the compiler to process code before compilation. Commands for that program, known as *directives*, are lines of the source file beginning with the character #, which distinguishes them from lines of source program text. The effect of each preprocessor directive is a change to the text of the source code, and the result is a new source code file, which does not contain the directives. The preprocessed source code, an intermediate file, must be a valid C program, because it becomes the input to the compiler.

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 197, which imbed files within the current file
- "Conditional compilation directives" on page 200, which conditionally compile sections of the current file
- "Message generation directives" on page 204, which control the generation of diagnostic messages
- ▶ IBM "Assertion directives (IBM extension)" on page 206, which specify attributes of the system the program is to run on
- "The null directive (#)" on page 207, which performs no action
- "Pragma directives" on page 207, which apply compiler-specific rules to specified sections of code

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" on page 190, which defines a macro
- "The #undef directive" on page 194, which removes a macro definition

"Standard predefined macro names" on page 196 describes the macros that are predefined by the ISO C standard. ▶ IBM Macros that are predefined for XL C are described in "Predefined macros" in the *XL C Compiler Reference*.

The #define directive

A *preprocessor define directive* directs the preprocessor to replace all subsequent occurrences of a macro with specified replacement tokens.

#define directive syntax
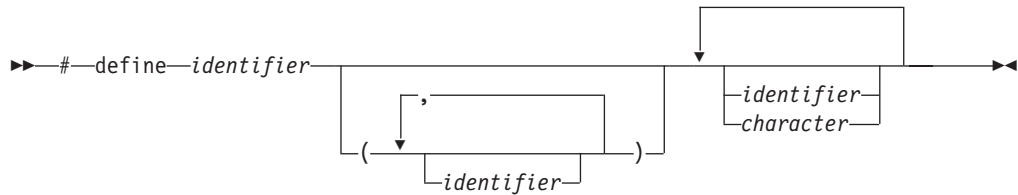
The `#define` directive can contain:
- "Object-like macros"
- "Function-like macros" on page 191

The following are some differences between `#define` and the `const` type qualifier:
- The `#define` directive can be used to create a name for a numerical, character, or string constant, whereas a `const` object of any type can be declared.
- A `const` object is subject to the scoping rules for variables, whereas a constant created using `#define` is not.
- Unlike a `const` object, the value of a macro does not appear in the intermediate source code used by the compiler because they are expanded inline. The inline expansion makes the macro value unavailable to the debugger.
- A macro can be used in a constant expression, such as an array bound, whereas a `const` object cannot.

Object-like macros

An *object-like macro definition* replaces a single identifier with the specified replacement tokens. The following object-like definition causes the preprocessor to replace all subsequent instances of the identifier COUNT with the constant 1000 :

```
#define COUNT 1000
```

If the statement

```
int arry[COUNT];
```

appears after this definition and in the same file as the definition, the preprocessor would change the statement to

```
int arry[1000];
```

in the output of the preprocessor.

Other definitions can make reference to the identifier COUNT:

```
#define MAX_COUNT COUNT + 100
```

The preprocessor replaces each subsequent occurrence of MAX_COUNT with COUNT + 100, which the preprocessor then replaces with 1000 + 100.

If a number that is partially built by a macro expansion is produced, the preprocessor does not consider the result to be a single value. For example, the following will not result in the value 10.2 but in a syntax error.

```
#define a 10
a.2
```

Identifiers that are partially built from a macro expansion may not be produced. Therefore, the following example contains two identifiers and results in a syntax error:

```
#define d efg
abcd
```

Function-like macros

More complex than object-like macros, a function-like macro definition declares the names of formal parameters within parentheses, separated by commas. An empty formal parameter list is legal: such a macro can be used to simulate a function that takes no arguments. 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 (...). In this case, the identifier __VA_ARGS__ may appear in the replacement list.

Function-like macro invocation:

An identifier followed by a comma-separated list of arguments in parentheses. The number of arguments should match the number of parameters in the macro definition, unless the parameter list in the definition ends with an ellipsis. In this latter case, the number of arguments in the invocation should exceed the number of parameters in the definition. The excess are called *trailing arguments*. Once the preprocessor identifies a function-like macro invocation, argument substitution takes place. A parameter in the replacement code is replaced by the corresponding argument. If trailing arguments are permitted by the macro definition, they are merged with the intervening commas to replace the identifier __VA_ARGS__, as if they were a single argument. Any macro invocations contained in the argument itself are completely replaced before the argument replaces its corresponding parameter in the replacement code.

A macro argument can be empty (consisting of zero preprocessing tokens). For example,

```
#define SUM(a,b,c) a + b + c
SUM(1,,3)  /* No error message.
              1 is substituted for a, 3 is substituted for c. */
```

If the identifier list does not end with an ellipsis, the number of arguments in a macro invocation must be the same as the number of parameters in the corresponding macro definition. During parameter substitution, any arguments remaining after all specified arguments have been substituted (including any separating commas) are combined into one argument called the variable argument.

The variable argument will replace any occurrence of the identifier __VA_ARGS__ in the replacement list. The following example illustrates this:

```
#define debug(...)   fprintf(stderr, __VA_ARGS__)

debug("flag");     /*   Becomes fprintf(stderr, "flag");   */
```

Commas in the macro invocation argument list do not act as argument separators when they are:

- In character constants
- In string literals
- Surrounded by parentheses

The following line defines the macro SUM as having two parameters a and b and the replacement tokens (a + b):

```
#define SUM(a,b) (a + b)
```

This definition would cause the preprocessor to change the following statements (if the statements appear after the previous definition):

```
c = SUM(x,y);
c = d * SUM(x,y);
```

In the output of the preprocessor, these statements would appear as:

```
c = (x + y);
c = d * (x + y);
```

Use parentheses to ensure correct evaluation of replacement text. For example, the definition:

```
#define SQR(c)  ((c) * (c))
```

requires parentheses around each parameter c in the definition in order to correctly evaluate an expression like:

```
y = SQR(a + b);
```

The preprocessor expands this statement to:

```
y = ((a + b) * (a + b));
```

Without parentheses in the definition, the correct order of evaluation is not preserved, and the preprocessor output is:

```
y = (a + b * a + b);
```

Arguments of the # and ## operators are converted *before* replacement of parameters in a function-like macro.

Once defined, a preprocessor identifier remains defined and in scope independent of the scoping rules of the language. The scope of a macro definition begins at the definition and does not end until a corresponding #undef directive is encountered. If there is no corresponding #undef directive, the scope of the macro definition lasts until the end of the translation unit.

A recursive macro is not fully expanded. For example, the definition

```
   #define x(a,b) x(a+1,b+1) + 4
```

expands

```
   x(20,10)
```

to

```
x(20+1,10+1) + 4
```

rather than trying to expand the macro x over and over within itself. After the macro x is expanded, it is a call to function x().

A definition is not required to specify replacement tokens. The following definition removes all instances of the token debug from subsequent lines in the current file:

```
#define debug
```

You can change the definition of a defined identifier or macro with a second preprocessor #define directive only if the second preprocessor #define directive is preceded by a preprocessor #undef directive. The #undef directive nullifies the first definition so that the same identifier can be used in a redefinition.

Within the text of the program, the preprocessor does not scan character constants or string constants for macro invocations.

The following example program contains two macro definitions and a macro invocation that refers to both of the defined macros:

```
/**
 ** This example illustrates #define directives.
 **/

#include <stdio.h>

#define SQR(s)  ((s) * (s))
#define PRNT(a,b) \
  printf("value 1 = %d\n", a); \
  printf("value 2 = %d\n", b) ;

int main(void)
{
  int x = 2;
  int y = 3;

    PRNT(SQR(x),y);

  return(0);
}
```

After being interpreted by the preprocessor, this program is replaced by code equivalent to the following:

```
#include <stdio.h>

int main(void)
{
  int x = 2;
  int y = 3;

    printf("value 1 = %d\n", ( (x) * (x) ) );
    printf("value 2 = %d\n", y);

  return(0);
}
```

This program produces the following output:

```
value 1 = 4
value 2 = 3
```

Variadic macro extensions (IBM extension)

Variadic macro extensions refer to two extensions to C99 related to macros with variable number of arguments. One extension is a mechanism for renaming the variable argument identifier from __VA_ARGS__ to a user-defined identifier. 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.

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.

> **Related reference**
>
> "The const type qualifier" on page 70
>
> "Operator precedence and associativity" on page 141
>
> "Parenthesized expressions ( )" on page 112
>
> "The ## operator" on page 196

The #undef directive

A *preprocessor undef directive* causes the preprocessor to end the scope of a preprocessor definition.

#undef directive syntax

►►—#—undef—*identifier*——————————————————————————◄◄

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 ## 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 190

Standard predefined macro names

C provides the following predefined macro names as specified in the ISO C language standard. Except for __FILE__ and __LINE__, the value of the predefined macros remains constant throughout the translation unit.

__DATE__

A character string literal containing the date when the source file was compiled.

The value of __DATE__ changes as the compiler processes any include files that are part of your source program. The date is in the form:

```
"Mmm dd yyyy"
```

where:

Mmm   Represents the month in an abbreviated form (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec).

dd   Represents the day. If the day is less than 10, the first d is a blank character.

yyyy   Represents the year.

__FILE__
A character string literal containing the name of the source file.

The value of __FILE__ changes as the compiler processes include files that are part of your source program. It can be set with the #line directive.

__LINE__
An integer representing the current source line number.

The value of __LINE__ changes during compilation as the compiler processes subsequent lines of your source program. It can be set with the #line directive.

__STDC__
For C, the integer 1 (one) indicates that the C compiler supports the ISO standard. If you set the language level to **classic**, 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.)

__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 compiled.

The value of __TIME__ changes as the compiler processes any include files that are part of your source program. The time is in the form:
```
"hh:mm:ss"
```

where:

hh   Represents the hour.

mm   Represents the minutes.

ss   Represents the seconds.

Related reference
"The #line directive" on page 205
Object-like macros

File inclusion directives

File inclusion directives consist of:
- "The #include directive" on page 198, which inserts text from another source file

The #include directive

A *preprocessor include directive* causes the preprocessor to replace the directive with the contents of the specified file.

#include directive syntax
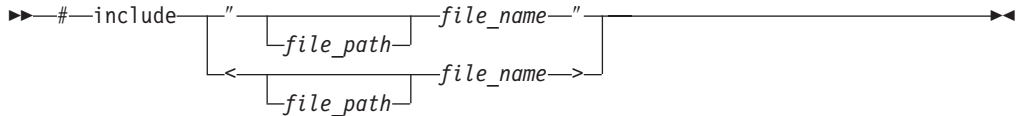
If the *file_name* is enclosed in double quotation marks, for example:

```
#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
int hour;
int min;
int sec;
#include "mydefs.h"
```

You can embed the definitions that appear in `defs.h` with the following directive:

```
#include "defs.h"
```

In the following example, a `#define` combines several preprocessor macros to define a macro that represents the name of the C standard I/O header file. A `#include` makes the header file available to the program.

```
#define C_IO_HEADER <stdio.h>

/* The following is equivalent to:
 *    #include <stdio.h>
 */

#include C_IO_HEADER
```

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

```
►►──#──include_next──┬─"─┬───────────┬──file_name──"─┬──────────►◄
                     │   └─file_path─┘                │
                     └─<─┬───────────┬──file_name─>───┘
                         └─file_path─┘
```
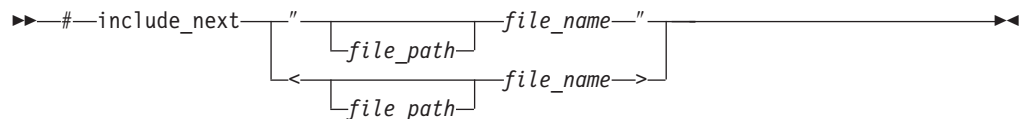
The directive must only be used in header files, and the file specified by the *file_name* must be a header file. There is no distinction between the use of double quotation marks and angle brackets to enclose the file name.

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 201, which conditionally include or suppress portions of source code, depending on the result of a constant expression
- "The #ifdef directive" on page 202, which conditionally includes source text if a macro name is defined
- "The #ifndef directive" on page 202, which conditionally includes source text if a macro name is not defined
- "The #else directive" on page 203, which conditionally includes source text if the previous `#if`, `#ifdef`, `#ifndef`, or `#elif` test fails
- "The #endif directive" on page 203, 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.

You can nest conditional compilation directives. In the following directives, the first `#else` is matched with the `#if` directive.

```
#ifdef MACNAME
                /*  tokens added if MACNAME is defined */
#   if TEST <=10
                /* tokens added if MACNAME is defined and TEST <= 10 */
#   else
                /* tokens added if MACNAME is defined and TEST >  10 */
```

```
#   endif
#else
                /* tokens added if MACNAME is not defined */

#endif
```

Each directive controls the block immediately following it. A block consists of all the tokens starting on the line following the directive and ending at the next conditional compilation directive at the same nesting level.

Each directive is processed in the order in which it is encountered. If an expression evaluates to zero, the block following the directive is ignored.

When a block following a preprocessor directive is to be ignored, the tokens are examined only to identify preprocessor directives within that block so that the conditional nesting level can be determined. All tokens other than the name of the directive are ignored.

Only the first block whose expression is nonzero is processed. The remaining blocks at that nesting level are ignored. If none of the blocks at that nesting level has been processed and there is a #else directive, the block following the #else directive is processed. If none of the blocks at that nesting level has been processed and there is no #else directive, the entire nesting level is ignored.

Related reference

"Message generation directives" on page 204

"Assertion directives (IBM extension)" on page 206

The #if and #elif directives

The #if and #elif directives compare the value of *constant_expression* to zero:

#if and #elif directive syntax
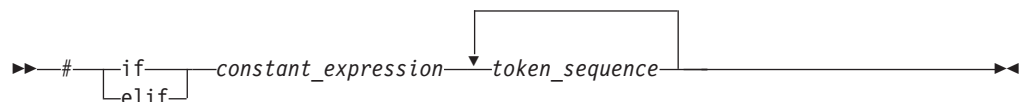
If the constant expression evaluates to a nonzero value, the lines of code that immediately follow the condition are passed on to the compiler.

If the expression evaluates to zero and the conditional compilation directive contains a preprocessor #elif directive, the source text located between the #elif and the next #elif or preprocessor #else directive is selected by the preprocessor to be passed on to the compiler. The #elif directive cannot appear after the preprocessor #else directive.

All macros are expanded, any defined() expressions are processed and all remaining identifiers are replaced with the token 0.

The *constant_expression* that is tested must be integer constant expressions with the following properties:
- No casts are performed.
- Arithmetic is performed using long int values.

- The *constant_expression* can contain defined macros. No other identifiers can appear in the expression.
- The *constant_expression* can contain the unary operator `defined`. This operator can be used only with the preprocessor keyword `#if` or `#elif`. The following expressions evaluate to 1 if the *identifier* is defined in the preprocessor, otherwise to 0:

```
defined identifier
defined(identifier)
```

For example:

```
#if defined(TEST1) || defined(TEST2)
```

Note: If a macro is not defined, a value of 0 (zero) is assigned to it. In the following example, TEST must be a macro identifier:

```
#if TEST >= 1
   printf("i = %d\n", i);
   printf("array[i] = %d\n", array[i]);
#elif TEST < 0
   printf("array subscript out of bounds \n");
#endif
```

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.

#ifdef directive syntax

```
►►──#──ifdef──identifier──┬──token_sequence──┬──newline_character──────────────►◄
                          └◄─────────────────┘
```
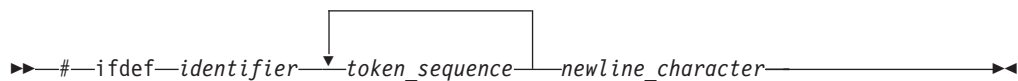
The following example defines MAX_LEN to be 75 if EXTENDED is defined for the preprocessor. Otherwise, MAX_LEN is defined to be 50.

```
#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

```
►►──#──ifndef──identifier──┬──token_sequence──┬──newline_character──────────────►◄
                           └◄─────────────────┘
```
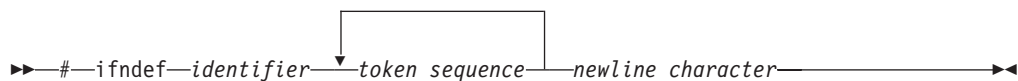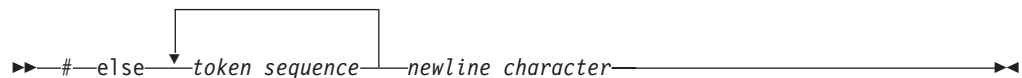
An identifier must follow the #ifndef keyword. The following example defines MAX_LEN to be 50 if EXTENDED is not defined for the preprocessor. Otherwise, MAX_LEN is defined to be 75.

```
#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

▶▶──#──else─┬─*token_sequence*─┬──*newline_character*──◀◀

The #endif directive

The preprocessor #endif directive ends the conditional compilation directive.

#endif directive syntax

▶▶──#──endif──*newline_character*──◀◀

Examples of conditional compilation directives

The following example shows how you can nest preprocessor conditional compilation directives:

```
#if defined(TARGET1)
#    define SIZEOF_INT 16
#    ifdef PHASE2
#        define MAX_PHASE 2
#    else
#        define MAX_PHASE 8
#    endif
#elif defined(TARGET2)
#    define SIZEOF_INT 32
#    define MAX_PHASE 16
#else
#    define SIZEOF_INT 32
#    define MAX_PHASE 32
#endif
```

The following program contains preprocessor conditional compilation directives:

```
/**
 ** This example contains preprocessor
 ** conditional compilation directives.
 **/

#include <stdio.h>
```

```
int main(void)
{
   static int array[ ] = { 1, 2, 3, 4, 5 };
   int i;

   for (i = 0; i <= 4; i++)
   {
      array[i] *= 2;

#if TEST >= 1
   printf("i = %d\n", i);
   printf("array[i] = %d\n",
   array[i]);
#endif

   }
   return(0);
}
```

Message generation directives

Message generation directives include the following:

- "The #error directive," which defines text for a compile-time error message
- "The #warning directive (IBM extension)" on page 205, which defines text for a compile-time warning message
- "The #line directive" on page 205, which supplies a line number for compiler messages

  Related reference

  "Conditional compilation directives" on page 200

The #error directive

A *preprocessor error directive* causes the preprocessor to generate an error message and causes the compilation to fail.

#error directive syntax

►►—#—error——*preprocessor_token*————————————————►◄

The #error directive is often used in the #else portion of a #if–#elif–#else construct, as a safety check during compilation. For example, #error directives in the source file can prevent code generation if a section of the program is reached that should be bypassed.

For example, the directive
```
#define BUFFER_SIZE 255

#if BUFFER_SIZE < 256
#error "BUFFER_SIZE is too small."
#endif
```

generates the error message:
```
BUFFER_SIZE is too small.
```

The #warning directive (IBM extension)

A *preprocessor warning directive* causes the preprocessor to generate a warning message but allows compilation to continue. The argument to #warning is not subject to macro expansion.

#warning directive syntax

```
►►──#──warning──┬─▼──preprocessor_token──┬─────────────────────────►◄
                └────────────────────────┘
```
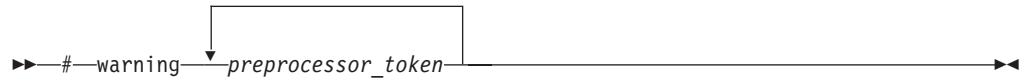
The preprocessor #warning directive is a language extension provided to facilitate handling programs developed with GNU C. The IBM implementation preserves multiple white spaces.

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

```
►►──#──line──┬─decimal_constant─┬────────────────────────┬──►◄
             │                  └─"──file_name──"─┘
             └─characters───────┘
```
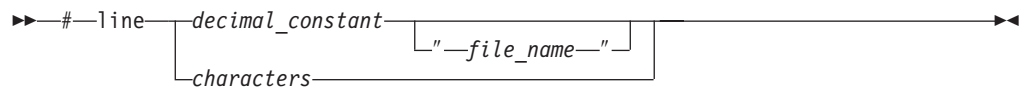
In order for the compiler to produce meaningful references to line numbers in preprocessed source, the preprocessor inserts #line directives where necessary (for example, at the beginning and after the end of included text).

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.

At the C99 language level, the maximum value of the #line preprocessing directive is 2147483647.

In all 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:

```
/**
 ** 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
```

> **Related reference**
>
> See __C99_MAX_LINE_NUMBER in the XL C Compiler Reference
>
> "Standard predefined macro names" on page 196

Assertion directives (IBM extension)

An *assertion directive* is an alternative to a macro definition, used to define the computer or system the compiled program will run on. Assertions are usually predefined, but you can define them with the #assert preprocessor directive.

#assert directive syntax

►►—#—assert—*predicate*—(—*answer*—)————————————————►◄

The *predicate* represents the assertion entity you are defining. The *answer* represents a value you are assigning to the assertion. You can make several assertions using the same predicate and different answers. All the answers for any given predicate are simultaneously true. For example, the following directives create assertions regarding font properties:

```
#assert font(arial)
#assert font(blue)
```

Once an assertion has been defined, the assertion predicate can be used in conditional directives to test the current system. The following directive tests whether arial or blue is asserted for font:

```
#if #font(arial) || #font(blue)
```

You can test whether any answer is asserted for a predicate by omitting the answer in the conditional:

```
#if #font
```

Assertions can be cancelled with the `#unassert` directive. If you use the same syntax as the `#assert` directive, the directive cancels only the answer you specify. For example, the following directive cancels the `arial` answer for the `font` predicate:

```
#unassert font(arial)
```

An entire predicate is cancelled by omitting the answer from the `#unassert` directive. The following directive cancels the `font` directive altogether:

```
#unassert font
```

Related reference

"Conditional compilation directives" on page 200

Predefined assertions

The following assertions are predefined for the AIX platform:

Table 30. Predefined assertions for AIX

#machine	#system(unix)	#cpu
	#system(aix)	

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 195

Pragma directives

A *pragma* is an implementation-defined instruction to the compiler. It has the general form:

#pragma directive syntax

```
►►─#─pragma─┬──────┬─┬─►──────────────────┬─new-line──────────►◄
            └─STDC─┘ └─◄character_sequence─┘
```
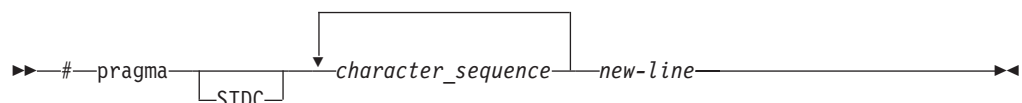
The *character_sequence* is a series of characters giving a specific compiler instruction and arguments, if any. The token STDC indicates a standard pragma; consequently,

no macro substitution takes place on the directive. The *new-line* character must terminate a pragma directive.

The *character_sequence* on a pragma is subject to macro substitutions. For example,

```
#define
XX_ISO_DATA
isolated_call(LG_ISO_DATA)
// ...
#pragma XX_ISO_DATA
```

Note: You can also use the _Pragma operator syntax to specify a pragma directive; for details, see "The _Pragma preprocessing operator."

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 XL C are described in "General purpose pragmas" in the *XL C Compiler Reference*.

The _Pragma preprocessing operator

The unary operator _Pragma, which is a C99 feature, allows a preprocessor macro to be contained in a pragma directive.

_Pragma operator syntax

```
►►── _Pragma──(──"──string_literal──"──)──────────────────────────────►◄
```

The *string_literal* may be prefixed with L, making it a wide-string literal.

The string literal is destringized and tokenized. The resulting sequence of tokens is processed as if it appeared in a pragma directive. For example:

```
_Pragma ( "pack(full)" )
```

would be equivalent to

```
#pragma pack(full)
```

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:

```
►►──#pragma──STDC──┬─FP_CONTRACT─────┬──┬─ON──────┬──new-line──────────►◄
                   ├─FENV_ACCESS─────┤  ├─OFF─────┤
                   └─CX_LIMITED_RANGE┘  └─DEFAULT─┘
```

The FP_CONTRACT and FENV_ACCESS pragmas are recognized and ignored.

CX_LIMITED_RANGE is described below.

pragma STDC CX_LIMITED_RANGE

The usual mathematical formulas for complex multiplication, division, and absolute value are problematic because of their treatment of infinities and because of undue overflow and underflow. The usual formulas are as follows:

$(x + iy) \times (u + iv) = (xu - yv) + i(yu + xv)$

$(x + iy)/(u + iv) = [(xu + yv) + i(yu - xv)]/(u^2 + v^2)$

$|\ x + iy\ | = \mathrm{sqrt}(x^2 + y^2)$

By default, the compiler uses slightly more complex but mathematically safer algorithms to implement these calculations. Where you determine that the usual mathematical formulas are safe, you can use the STDC CX_LIMITED_RANGE pragma to inform the compiler that, when the state is "on", the formulas are acceptable. In doing so, you allow the compiler to generate faster code for these computations. When the state is "off", the compiler will continue to use the safer algorithms. For details on the implementation of this pragma, see **#pragma STDC cx_limited_range** in the *XL C Compiler Reference*.

Chapter 11. The IBM XL C language extensions

The IBM XL C extensions include C features as extensions to C89, Unicode support, GNU C compatibility, vector processing support, and decimal floating-point support.

C99 features as extensions to C89

The following features are enabled by default when you compile with any of the following:

- the **xlc** invocation command
- the **c99** invocation command
- the **-qlanglvl=extc99 | stcd99 | extc89 | extended** options

For more information on these options, see the **-qlanglvl** option in the *XL C Compiler Reference*.

Table 31. Default C99 features as extensions to C89

Language feature	Discussed in:
Hexadecimal floating-point constants	Hexadecimal floating-point literals
__func__ predefined identifier	"The __func__ predefined identifier" on page 13
Concatenation of wide and non-wide character strings	String concatenation
Mixed declarations and code	"Overview of data declarations and definitions" on page 39
long long data type	"Integral types" on page 47
Complex data type	"Complex floating-point types " on page 50
_Bool data type	"Boolean types" on page 48
Trailing comma allowed in enum declaration	"Enumeration type definition" on page 62
Duplicate type qualifiers	"Type qualifiers" on page 67
Variable length arrays	"Variable length arrays" on page 84
Non-lvalue array subscripts	"Array subscripting operator [ ]" on page 133
Flexible array members at the end of a structure or union	Flexible array members
Non-constant expression in initializer for structure or union	"Initialization of structures and unions" on page 90
Designated initializers	"Designated initializers for aggregate types (C only)" on page 87
Removal of implicit function declaration	"Function declarations" on page 170
Removal of implicit int return type in function declarations	"Function return type specifiers" on page 176
Static arrays as function parameters	"Static array indices in function parameter declarations (C only)" on page 178
Variable arguments in function-like macros	"Function-like macros" on page 191

Table 31. Default C99 features as extensions to C89 (continued)

Language feature	Discussed in:
Empty arguments in function-like macros	"Function-like macros" on page 191
Additional predefined macro names	"Standard predefined macro names" on page 196
Compound literals	"Compound literal expressions" on page 140
_Pragma operator	"The _Pragma preprocessing operator" on page 208
Standard pragmas	"Standard pragmas" on page 208
New limit for #line directive	"The #line directive" on page 205

The following features are enabled by default when you compile with any of the following:

- the **xlc** invocation command
- the **c99** invocation command
- the **-qlanglvl=extc99 | stcd99 | extc89 | extended** options

They are also enabled or disabled by specific compiler options, which are listed in the table below.

Table 32. Default C99 features as extensions to C89, with individual option controls

Language feature	Discussed in:	Individual option control
Digraphs	"Digraph characters" on page 32	**-q[no]digraph**
C++ style comments	"Comments" on page 33	**-q[no]cpluscmt**
The inline function specifier	"The inline function specifier" on page 173	**-qkeyword=inline**

The following feature is enabled by default when you compile with any of the following:

- the **xlc** invocation command
- the **c99** invocation command
- the **-qlanglvl=extc99 | stcd99** options

Language feature	Discussed in:
Unsuffixed long long integer literals	Decimal integer literals

The following features are enabled by default when you compile with any of the following:

- the **xlc** invocation command
- the **c99** invocation command
- the **-qlanglvl=extc99 | stcd99** options

They are also enabled or disabled by specific compiler options, which are listed in the table below.

Table 33. Strict C99 features as extensions to C89, with individual option controls

Language feature	Discussed in:	Individual option control
Universal character names	"The Unicode standard" on page 30	**-qlanglvl=[no]ucs**
The restrict type qualifier	"The restrict type qualifier" on page 70	**-qkeyword=restrict**

Related reference

📄 See -qpluscmt in the XL C Compiler Reference

📄 See -qkeyword in the XL C Compiler Reference

📄 See -qdigraph in the XL C Compiler Reference

📄 See Invoking the compiler in the XL C Compiler Reference

Extensions for Unicode support

The following feature requires compilation with the use of an additional option.

Language feature	Discussed in:	Required compilation option
UTF-16, UTF-32 literals	"UTF literals (IBM extension)" on page 31	**-qutf**

Related reference

📄 See -qutf in the XL C Compiler Reference

Extensions for GNU C compatibility

The following feature is enabled by default at all language levels.

Table 34. Default IBM XL C extensions for GNU C compatibility

Language feature	Discussed in:
#include_next preprocessor directive	"The #include_next directive (IBM extension)" on page 199

The following features are enabled by default when you compile with any of the following:
- the **xlc** invocation command
- the **-qlanglvl=extc99 | extc89 | extended** options

Table 35. Default IBM XL C extensions for GNU C compatibility

Language feature	Discussed in:
Alternate keywords	"Keywords for language extensions (IBM extension)" on page 12
__extension__ keyword	"Keywords for language extensions (IBM extension)" on page 12

Table 35. Default IBM XL C extensions for GNU C compatibility (continued)

Language feature	Discussed in:
asm labels	"Assembly labels (IBM extension)" on page 14
Complex literal suffixes	Complex literals
Global register variables	"Variables in specified registers (IBM extension)" on page 44
Placement of flexible array members anywhere in structure or union	Flexible array members
Static initialization of flexible array members of aggregates	Flexible array members
Zero-extent arrays	Zero-extent array members (IBM extension)
Type attributes	"Type attributes (IBM extension)" on page 72
Variable attributes	"Variable attributes (IBM extension)" on page 96
Locally declared labels	"Locally declared labels (IBM extension)" on page 146
Labels as values	"Labels as values (IBM extension)" on page 146
__alignof__ operator	"The __alignof__ operator (IBM extension)" on page 119
__typeof__ operator	"The typeof operator (IBM extension)" on page 121
Generalized lvalues	"Lvalues and rvalues" on page 109
Complex type arguments to unary operators	"Unary expressions" on page 115
Initialization of static variables by compound literals	"Compound literal expressions" on page 140
__imag__ and __real__ complex type operators	"The __real__ and __imag__ operators (IBM extension)" on page 122
Cast to a union type	"Cast to union type (C only) (IBM extension)" on page 138
Computed goto statements	"Computed goto statement (IBM extension)" on page 162
Statements and declarations in expressions	"Statement expressions (IBM extension)" on page 148
Function attributes	"Function attributes (IBM extension)" on page 179
__inline__ function specifier	"The inline function specifier" on page 173
Nested functions	"Nested functions (IBM extension)" on page 188
Variadic macro extensions	Variadic macro extensions(IBM extension)
#warning preprocessor directive	"The #warning directive (IBM extension)" on page 205
#assert, #unassert preprocessor directives	"Assertion directives (IBM extension)" on page 206

The following features are enabled by default when you compile with any of the following:
- the **xlc** invocation command
- the **-qlanglvl=extc99 | extc89 | extended** options

They are also enabled or disabled by specific compiler options, which are listed in the table below.

Table 36. IBM XL C extensions for GNU C compatibility with individual option controls

Language feature	Discussed in:	Individual option controls
`asm`, and `__asm` keywords	"Assembly labels (IBM extension)" on page 14, "Inline assembly statements (IBM extension)" on page 163	**-qkeyword=asm**, **-qasm**
`asm` inline assembly-language statements	"Inline assembly statements (IBM extension)" on page 163	**-qasm**

The following features require compilation with the use of an additional option.

Table 37. IBM XL C extensions for GNU C compatibility, requiring additional compiler options

Language feature	Discussed in:	Required compilation option
Dollar signs in identifiers	"Characters in identifiers" on page 13	**-qdollar**
The `typeof` keyword	"The typeof operator (IBM extension)" on page 121	**-qkeyword=typeof**
The `__thread` storage class specifier	"The __thread storage class specifier (IBM extension)" on page 46	**-qtls**

Related reference

See -qkeyword in the XL C Compiler Reference

See -qasm in the XL C Compiler Reference

See -qtls in the XL C Compiler Reference

See -qdollar in the XL C Compiler Reference

See Invoking the compiler in the XL C Compiler Reference

Extensions for vector processing support

The vector extensions are only accepted when all of the following conditions are met:

1. The **-qarch** option is set to a target architecture that supports vector processing instructions.
2. The **-qenablevmx** option is in effect.
3. The **-qaltivec** option is in effect.

For more information on these options, see the *XL C Compiler Reference*.

Table 38. IBM XL C extensions to support the AltiVec Application Programming Interface specification

Language feature	Discussed in:
Vector programming language extensions	"Vector types (IBM extension)" on page 52, "Vector literals (IBM extension)" on page 22

The following features are IBM extensions to the AltiVec Application Programming Interface specification.

Table 39. IBM XL C extensions to the AltiVec Application Programming Interface specification

Language extension	Discussed in:
Initializer lists for vector constants	"Initialization of vectors (IBM extension)" on page 89
typedef definitions for vector types	"typedef definitions" on page 65
compound literals as initializers for static vector variables	"Compound literal expressions" on page 140
vector types as arguments to the __alignof__ and typeof operators	"The __alignof__ operator (IBM extension)" on page 119, "The typeof operator (IBM extension)" on page 121

Extensions for decimal floating-point support

The following feature requires compilation with the use of an additional option.

Language feature	Discussed in:	Required compilation option
Decimal floating-point types	"Real floating-point types" on page 49, Decimal floating-point literals (IBM extension), "Floating-point conversions" on page 102	-qdfp

Notices

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

COPYRIGHT LICENSE:

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

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

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1998, 2008. All rights reserved.

Trademarks and service marks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol ($^{®}$ or $^{™}$), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A complete and current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

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

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

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.

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

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

Index

Special characters

__align   68
__cdecl   187
__func__   12
__VA_ARGS__   190
_Pragma   208
_thread storage class specifier   46
- (subtraction operator)   127
- (unary minus operator)   117
-- (decrement operator)   116
-> (arrow operator)   114
, (comma operator)   134
! (logical negation operator)   117
!= (not equal to operator)   129
? : (conditional operators)   136
/ (division operator)   126
/= (compound assignment
  operator)   124
. (dot operator)   114
$ 12, 28
* (indirection operator)   118
* (multiplication operator)   126
*= (compound assignment operator)   124
\ continuation character   15, 189
\ escape character   29
[ ] (array subscript operator)   133
% (remainder)   126
> (greater than operator)   128
>> (right-shift operator)   127
>>= (compound assignment
  operator)   124
>= (greater than or equal to
  operator)   128
< (less than operator)   128
<< (left-shift operator)   127
<<= (compound assignment
  operator)   124
<= (less than or equal to operator)   128
| (bitwise inclusive OR operator)   131
| (vertical bar), locale   28
|| (logical OR operator)   132
& (address operator)   118
& (bitwise AND operator)   130
&& (label value operator)   140, 146
&& (logical AND operator)   132
&= (compound assignment
  operator)   124
preprocessor directive character   189
preprocessor operator   195
(macro concatenation)   196
+ (addition operator)   127
+ (unary plus operator)   117
++ (increment operator)   115
+= (compound assignment
  operator)   124
= (simple assignment operator)   124
== (equal to operator)   129
^ (bitwise exclusive OR operator)   131
^ (caret), locale   28
^= (compound assignment
  operator)   124

~ (bitwise negation operator)   117

A

addition operator (+)   127
address operator (&)   118
  GNU C extension   146
aggregate types   37
  initialization   90
alias
  function   12
  type-based aliasing   81
alias function attribute   180
alignment   97, 98
  bit fields   55
  structure members   55
  structures   97
  structures and unions   68
alignof operator   119
always_inline function attribute   181
AND operator, bitwise (&)   130
AND operator, logical (&&)   132
argc (argument count)   184
  example   184
arguments
  macro   190
  main function   184
  passing   169, 185
  passing by reference   186
  passing by value   186
  trailing   190
argv (argument vector)   184
  example   184
arithmetic conversions   101
arithmetic types
  type compatibility   51
arrays
  array-to-pointer conversions   105
  as function parameter   42, 177
  declaration   42, 177
  description   83
  flexible array member   55
  initialization   87
  initializing   93
  multidimensional   83
  subscripting operator   133
  type compatibility   85
  variable length   78, 84
  zero-extent   55
ASCII character codes   29
asm   11
  keyword   12, 44
  labels   12
  statements   163
assembly
  labels   12
  statements   163
assignment operator (=)
  compound   124
  pointers   82
  simple   124

B

basic example, described   x
binary expressions and operators   123
bit fields   55
  as structure member   55
  type name   121
bitwise negation operator (~)   117
block statement   147
block visibility   4
bool   52
Boolean
  conversions   102
  data types   48
  literals   15
break statement   158
built-in data types   37

C

C99 long long under C++
  promotion   104
case label   150
cast expressions   15, 52, 138
  union type   138
char type specifier   50
character
  data types   50
  literals   15
  multibyte   15, 28
character set
  extended   28
  source   28
class members
  access operators   114
classes
  class objects   37
comma   134
  in enumerator list   62
comments   33
compatibility
  C89 and C99   211
  data types   37
  user-defined types   64
  XL C and GCC   1, 213
  XL C++ and C decimal-floating point
    types   1
  XL C++ and C99   1
compatible functions   172
compatible types
  across source files   64
  arithmetic types   51
  arrays   85
  in conditional expressions   136
complex types   49
composite types   37
  across source files   64

variable length array   37, 84, 161
   as function parameter   84, 185
   sizeof   112
   type name   78
variably modified types   83, 84, 150
vector data types   52
vector processing support   1, 215
vector types   121
   in typedef declarations   65
   literals   15
visibility   3
   block   4
void   51
   in function definition   176, 177
   pointer   105, 106
volatile
   qualifier   67, 71

W

warning preprocessor directive   205
wchar_t type specifier   15, 47, 50
weak symbol   99
while statement   154
white space   11, 33, 189, 195
wide characters
   literals   15
wide string literal   15

Z

zero-extent array   55

IBM ®

Program Number: 5724-U80

Printed in USA