XL C Enterprise Edition for AIX

XL C Language Reference

Version 7.0

SC09-7895-00

XL C Enterprise Edition for AIX

XL C Language Reference

Version 7.0

SC09-7895-00

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 183.

First Edition (September, 2004)

This edition applies to Version 7.0.0 of IBM XL C Enterprise Edition for AIX® (product number 5724-I10) and to all subsequent releases and modifications until otherwise indicated in new editions.

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

`compinfo@ca.ibm.com`

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

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

Contents

About This Reference

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

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

To avoid possible ambiguity and confusion with K&R C, this reference uses the term *Classic C* to refer to the C language plus the generally accepted extensions produced by Brian Kernighan and Dennis Ritchie (K&R C) that were in use prior to the ISO standardization of C.

The focus of this book is on the fundamentals and intricacies of the C language. The availability of a particular language feature at a particular language level is controlled by compiler options. Comprehensive coverage of the possibilities offered by the compiler options is available in *XL C Compiler Reference*.

The C language described in this reference is based on the following standards:
- The C language described in *Programming languages – C* (ISO/IEC 9899:1990), henceforth referred to as *C89*. This was the first ISO C standard.
- The C language described in *Programming languages – C* (ISO/IEC 9899:1999), henceforth referred to as *C99*. This is an update to the C89 standard.

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

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

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

The IBM Language Extensions

Based on various language standards, this release contains language extensions that enhance usability and facilitate porting programs to different platforms.

We refer to the following language specifications as ″base language levels″ in order to introduce the notion of an extension to a base.
- C99
- C89

In addition, we also use *Classic C* to refer to the *de facto* K&R industry standard, which was commonly used by C implementations before C89 was standardized.

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

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

The language levels for C99 and C89 specify strict conformance. Classic C generally follows the *de facto* K&R industry standard. These language levels can be selected using the `-qlanglvl` compiler option. Extensions to the standard levels can also be specified using this option. For example, `-qlanglvl=stdc99` specifies the Standard C99, and `-qlanglvl=extc99` specifies C99 plus the orthogonal extensions. Refer to *XL C Compiler Reference* for details about `-qlanglvl` and its suboptions.

Any previously existing options continue to be supported, such as the compiler options `digraph`, `UCS` character, `long long`, and `dollar`, which are orthogonal extensions to C89.

Features Related to GNU C

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

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

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

Highlighting Conventions

Bold	Identifies commands, keywords, and other items whose names are predefined by the system.
Italics	Identify parameters whose actual names or values are to be supplied by the programmer. *Italics* are also used for the first mention of new terms.
`Example`	Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code, messages from the system, or information that you should actually type.

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

How to Read the Syntax Diagrams

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

  The ▶▶─── symbol indicates the beginning of a command, directive, or statement.

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

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

  The ───▶◀ symbol indicates the end of a command, directive, or statement.

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

  Note: In the following diagrams, `statement` represents a C command, directive, or statement.
- Required items appear on the horizontal line (the main path).

```
▶▶──statement──required_item──────────────────────────────────────────▶◀
```

- Optional items appear below the main path.

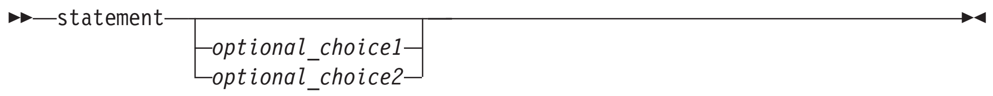
```
▶▶──statement──┬───────────────┬──────────────────────────────────────▶◀
               └─optional_item─┘
```

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

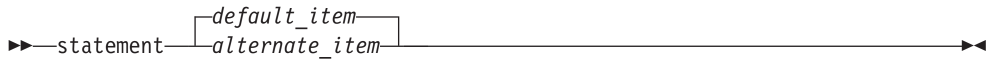
  If you *must* choose one of the items, one item of the stack appears on the main path.

```
▶▶──statement──┬─required_choice1─┬───────────────────────────────────▶◀
               └─required_choice2─┘
```

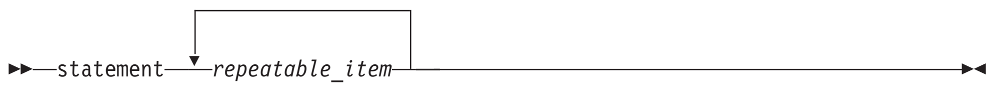
  If choosing one of the items is optional, the entire stack appears below the main path.



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



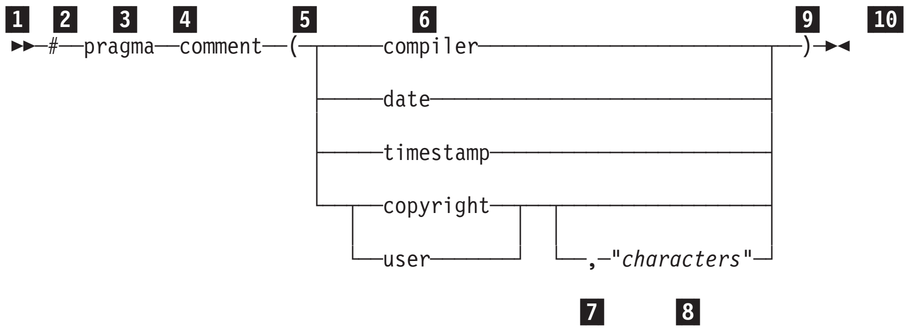
- An arrow returning to the left above the main line indicates an item that can be repeated.



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

  Variables appear in italicized lowercase letters (for example, *identifier*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

The following syntax diagram example shows the syntax for the **#pragma comment** directive. See "Pragma Directives (#pragma)" on page 175 for information on the **#pragma** directive.



1 This is the start of the syntax diagram.

2 The symbol # must appear first.

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

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

5 An opening parenthesis must be present.

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

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

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

9 A closing parenthesis is required.

10 This is the end of the syntax diagram.

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

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

Reading the Syntax Diagrams

Chapter 1. Scope and Linkage

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

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

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

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

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

Related References
- "Program Linkage" on page 4

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.

Differences in terminology between C and C++

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

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

```
int x;
void f() {

  int x = x;
}
```

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

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

Related References
- “Block Statement” on page 143

Function Scope

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

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

Related References
- "Labels" on page 141

Function Prototype Scope

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

Related References
- "Function Declarations" on page 121

Global Scope

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

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

Related References
- "Internal Linkage" on page 4

Name Spaces of Identifiers

Name spaces are the various syntactic contexts within which an identifier can be used. Within the same context and the same scope, an identifier must uniquely identify an entity. The compiler sets up *name spaces* to distinguish among identifiers referring to different kinds of entities. Identical identifiers in different name spaces do not interfere with each other, even if they are in the same scope.

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

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

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

- Enumeration constants
- `typedef` names.

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

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

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

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

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

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

Program Linkage

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

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

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

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

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

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

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

No Linkage

The following kinds of identifiers have no linkage:

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

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

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

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

Chapter 2. Lexical Elements

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

Tokens

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

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

Punctuators

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

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

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

or any of the following:

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

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

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

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

```
*=        /=        %=        +=        -=
<<=       >>=       &=        ^=        |=
<:        :>        <%        %>        %:        %:%:
```

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

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

Alternative Tokens

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

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

Source Program Character Set

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

- The uppercase and lowercase letters of the English alphabet
```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```
- The decimal digits `0` through `9`
```
0 1 2 3 4 5 6 7 8 9
```
- The following graphic characters:
```
! " # % & ' ( ) * + , - . / :
; < = > ? [ \ ] _ { } ~
```
  – The caret (^) character in ASCII (bitwise exclusive OR symbol).
  – The split vertical bar (¦) character in ASCII.
- The space character
- The control characters representing new-line, horizontal tab, vertical tab, and form feed, and end of string (`NULL` character)

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

Escape Sequences

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

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

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

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

The escape sequences and the characters they represent are:

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

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

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

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

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

This statement results in the following output:

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

The Unicode Standard

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

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

C99 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**
`\UNNNNNNNN`	`NNNNNNNN`
`\uNNNN`	`0000NNNN`
where `N` *is a hexadecimal digit*	

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.

XL C/C++ implements the data types **uint_least16_t** and **uint_least32_t** to process UTF-16 and UTF-32 characters in C and C++ in conformance with the Unicode Standard. The data types, also referred to as *u-literals* and *U-literals*, respectively, are the string literals required by the Unicode Standard to specify a UTF-16 or UTF-32 character, and were approved by the C Standards Committee. Previously, a UTF-16 character was represented by an **unsigned short**, and a UTF-32 character, by an **unsigned int**.

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

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

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

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

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

For example,

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

String concatenation

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

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

Multiple concatentations are allowed, with these rules applied recursively.

Trigraph Sequences

Some characters from the C 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.

Multibyte Characters

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

Related References
- "char and wchar_t Type Specifiers" on page 41

Comments

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

A comment consists of the /* (slash, asterisk) characters, followed by any sequence of characters (including new lines), followed by the */ characters.

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

Multibyte characters can also be included within a comment.

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

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

```
 #include <stdio.h>

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

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

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

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

```
#include <stdio.h>

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

The output of the program is:

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

In the following example, the comments are highlighted:

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

   #include <stdio.h>

   int main(void)
```

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

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

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

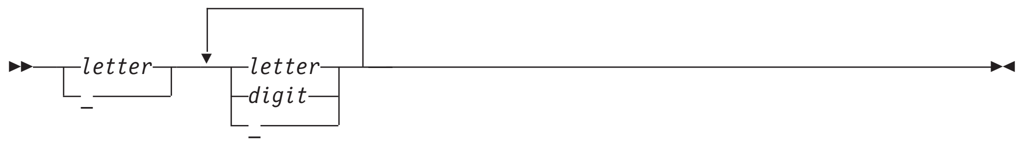
```
    /* A program with conditional compilation to avoid nested comments.
*/
    #define TEST_FUNCTION 0
    #include <stdio.h>

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

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

Related References
- "Trigraph Sequences" on page 11

Identifiers

Identifiers provide names for the following language elements:
- Functions
- Objects
- Labels
- Function parameters
- Macros and macro parameters
- Typedefs
- Enumerated types and enumerators
- Struct and union names

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



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

Reserved Identifiers

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

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

Case Sensitivity and Special Characters in Identifiers

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

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

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

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

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

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

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

Predefined Identifiers

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

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

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

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

Keywords

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

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

The C language also reserves the following keywords:

restrict	**_Bool**	**_Complex**	**uint_least16_t**
		_Imaginary	**uint_least32_t**

Keywords for Language Extensions

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

typeof	**__attribute__**	**__imag__**	**__restrict**
__align	**__complex__**	**__inline__**	**__signed__**
__alignof__	**__const__**	**__label__**	**__typeof__**
__asm	**__extension__**	**__real__**	**__volatile__**
__asm__			

Alternative Representations of Operators and Punctuators

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

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

Literals

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

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

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

Boolean Literals

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

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

Related References
- "Boolean Variables" on page 41
- "Lvalues and Rvalues" on page 86

Integer Literals

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

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

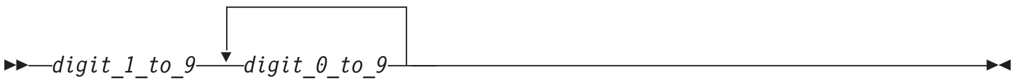
The data type of an integer literal is determined by its form, value, and suffix. The following table lists the integer literals and shows the possible data types. The smallest data type that can represent the constant value is used to store the constant.

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

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

Decimal Integer Literals

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



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

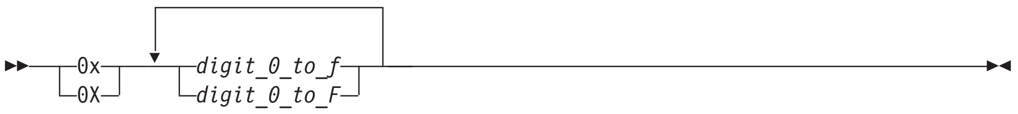
The following are examples of decimal literals:

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

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

Hexadecimal Integer Literals

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

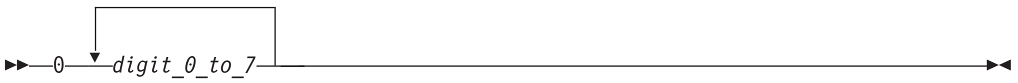


The following are examples of hexadecimal integer literals:

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

Octal Integer Literals

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



The following are examples of octal integer literals:

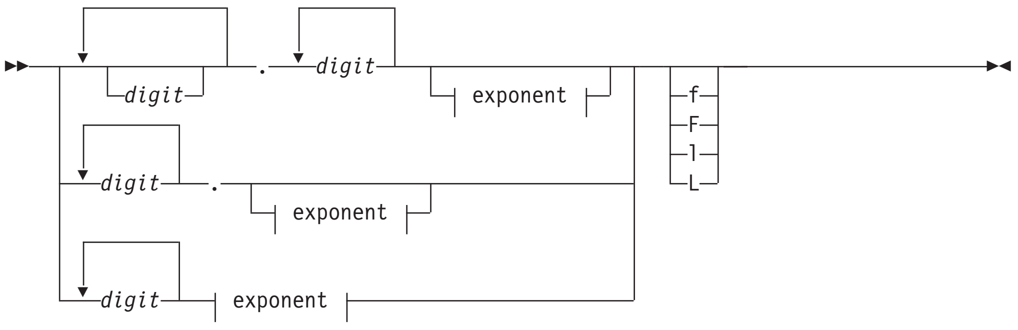
```
0
0125
034673
03245
```

Floating-Point Literals

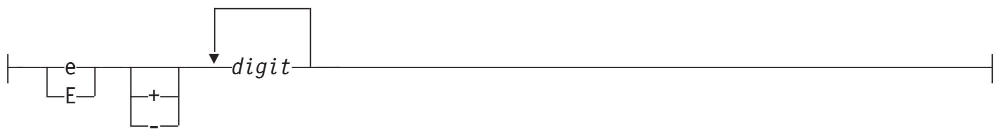
A *floating-point literal* consists of the following:

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

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



Exponent:



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

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

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

The following are examples of floating-point literals:

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

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

Related References
- "Floating-Point Variables" on page 42
- "Unary Expressions" on page 94

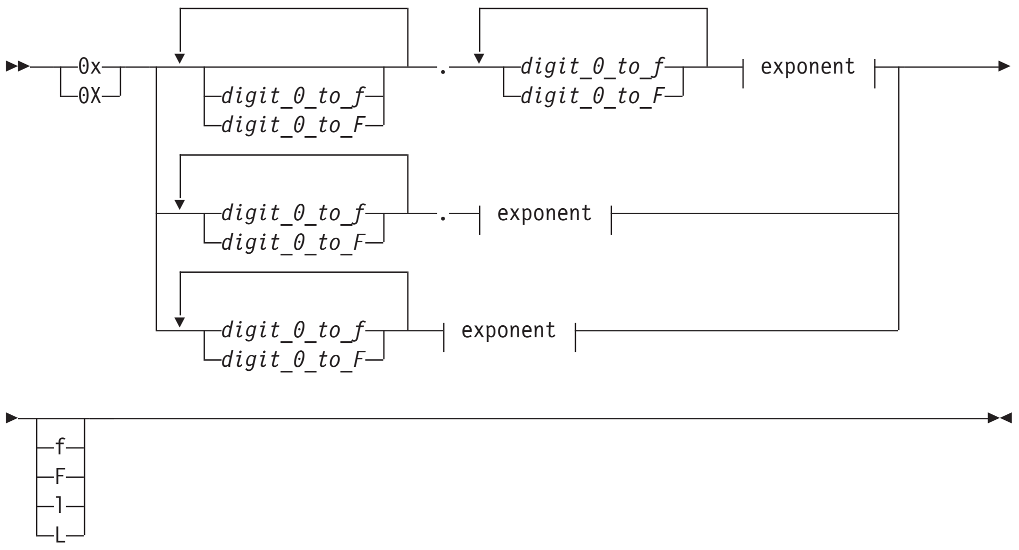
Hexadecimal Floating Constants

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

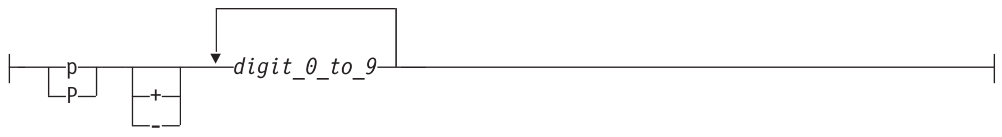
The significant part represents a rational number and is composed of the following:
- a sequence of hexadecimal digits (whole-number part)
- an optional fraction part

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:



Exponent:



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

Complex Literals

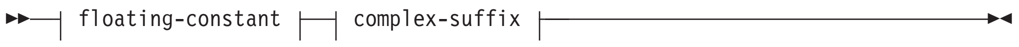
A complex literal type represents a complex number. The predefined macro _Complex_I represents a constant expression of type const float _Complex with the value of the imaginary unit. For example,

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

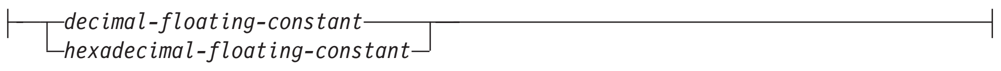
initializes the variable varComplex to type float _Complex.

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

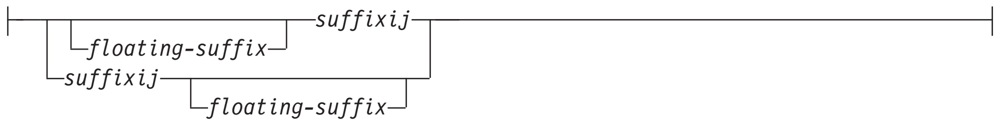
The simplified syntax for a complex literal is:



floating-constant:



complex-suffix:



where

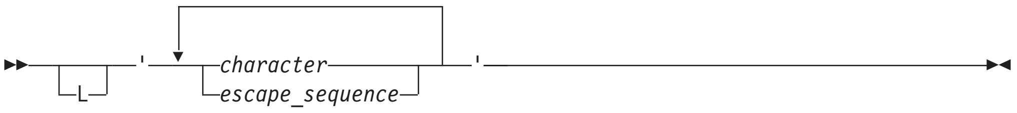
floating-suffix	is one of `f`, `F`, `l` (lowercase *el*) or `L`. The suffixes `f` or `F` indicates a complex literal of type `float _Complex`. The suffixes `l` or `L` indicates a complex literal of type `long double _Complex`. A complex literal is of type `double _Complex` in the absense of suffixes.
suffixij	is one of `i`, `I`, `j`, `J`.

Related References
- "Unary operators for complex types" on page 94

Character Literals

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

Character literals have the following form:



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

A character literal has type **int**.

A wide character literal has type **wchar_t**, and a multicharacter literal has type **int**.

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

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

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

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

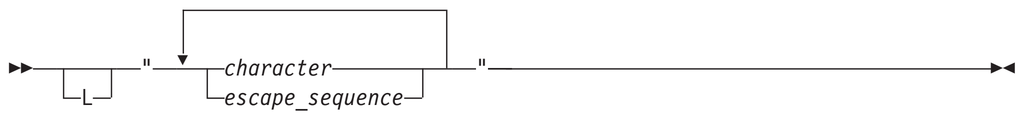
The following are examples of character literals:

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

Related References
- "char and wchar_t Type Specifiers" on page 41
- "The Unicode Standard" on page 10

String Literals

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



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

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

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

The following are examples of string literals:

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

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

Multiple spaces contained within a string literal are retained.

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

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

Concatenation

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

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

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

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

Following any concatenation, '\0' of type **char** is appended at the end of each string. For a wide string literal, '\0' of type **wchar_t** is appended. For example:

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

Compound Literals

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

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

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

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

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

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

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

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

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

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

Chapter 3. Declarations

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

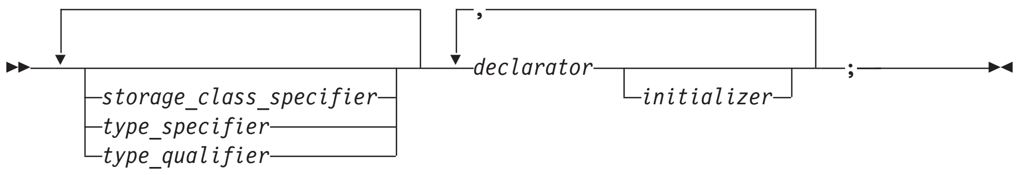
In diverse ways, declarations determine the interrelated attributes of an object: storage class, type, scope, visibility, storage duration, and linkage.

Declaration Overview

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

The lexical order of elements of a declaration for a data object is as follows:
- Storage duration and linkage specification
- Type specification
- Declarators, which introduce identifiers and make use of type qualifiers and storage qualifiers
- Initializers, which initialize storage with initial values

All data declarations have the form:



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

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

Declarations	Declarations and Definitions

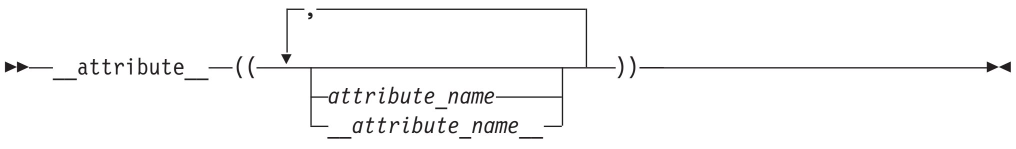
`struct payroll;`	`struct payroll {`
	`        char *name;`
	`        float salary;`
	`     } employee;`

Related References
- Chapter 4, "Declarators," on page 67

Variable Attributes

Variable attributes are orthogonal language extensions provided to facilitate handling programs developed with the GNU C compiler. These language features allow you use named attributes to modify the declarations of variables. The syntax and supported variable attributes are described in this section. For unsupported attribute names, the XL C issues diagnostics and ignores the attribute specification.

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



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

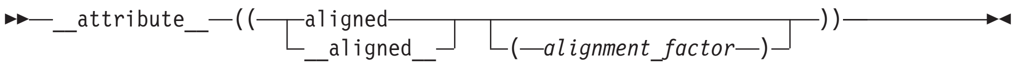
A variable attribute specification using the form *__attribute_name__* (that is, the variable attribute keyword with double underscore characters leading and trailing) reduces the likelihood of a name conflict with a macro of the same name.

Related References
- "Function Attributes" on page 123
- "Type Attributes" on page 36

The aligned Variable Attribute

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

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



where *alignment_factor* is a constant expression that evaluates to 1 or a positive power of two.

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

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

The mode Variable Attribute

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

Valid arguments for attribute **mode** are `byte`, `word`, and `pointer`, and the forms of these modes with leading and trailing double underscores.
- `byte` means a 1-byte integer type
- `word` means a 4-byte integer type
- `pointer` means 4-byte integer type in 32-bit mode and an 8-byte integer type in 64-bit mode

The syntax is as follows:

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

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

The packed Variable Attribute

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

The syntax is as follows:

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

The weak Variable Attribute

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

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

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

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

The __align Specifier

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

A declaration takes one of the following forms:

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

Structure or union syntax:

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

where:

int_constant
: Is a positive integer value indicating the byte-alignment boundary. The legal values are 1, or any positive power of two.

struct_or_union_specifier
: Is a structure or union specifier.

struct_declaration_list
: Is a structure declaration list.

tag
: Is a structure or union identifier.

Restrictions and limitations

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

Not all alignments may be representable in an object file.

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

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

Tentative Definitions

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

The following statements show normal definitions and tentative definitions.

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

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

Related References
- “Declaration Overview” on page 25
- “Storage Class Specifiers”

Objects

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

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

The C language has built-in, or *fundamental*, data types and user-defined data types. Standard data types include signed and unsigned integers, floating-point numbers, and characters. Enumerations, structures, and unions are considered user-defined types.

Related References
- “Lvalues and Rvalues” on page 86

Storage Class Specifiers

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

For a variable, its default storage duration, scope, and linkage depend on where it is declared: whether inside or outside a block statement or the body of a function. When these defaults are not satisfactory, you can specify an explicit storage class: **auto**, **static**, **extern**, or **register**.

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

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

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

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

The only storage class specifiers allowed in a namespace or global scope declaration are **static** and **extern**.

The storage class specifiers are:
- **auto**
- **extern**
- **register**
- **static**
- **typedef**

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

auto Storage Class Specifier

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

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

Initialization

You can initialize any **auto** variable except 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.

Storage duration

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

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

Linkage

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

Related References
- "Block Statement" on page 143
- "goto Statement" on page 156
- "Function Declarations" on page 121

extern Storage Class Specifier

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

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

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

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

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

Related References
- "The extern inline keyword" on page 138

Initialization

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.

Storage duration

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

Linkage

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

Related References
- "External Linkage" on page 5
- "Internal Linkage" on page 4
- "static Storage Class Specifier" on page 33
- "Inline Functions" on page 137

register Storage Class Specifier

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

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

Initialization

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

Storage duration

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

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

Linkage

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

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

static Storage Class Specifier

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

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

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

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

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

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.

Initialization

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

More precisely, in C,

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

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

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

Linkage

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

Example

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

```
#include <stdio.h>

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

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

The following is the output of the above example:

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

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

typedef

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

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

Examples of typedef Declarations

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

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

The following declarations are equivalent to the above declaration:

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

Similarly, **typedef** can be used to define a class type (structure 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.

Type Specifiers

Type specifiers indicate the type of the object or function being declared. The following are the available kinds of type specifiers:
- Simple type specifiers
- Enumerated specifiers
- **const** and **volatile** qualifiers

The term *scalar types* collectively refers in C to arithmetic types or pointer types.

The term *aggregate type* refers to array and structure types.

Type Names

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

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

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

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

The following constructions illustrate applications of the type naming rules.

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

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

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

Type Attributes

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

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

```
▶▶──__attribute__──((──┬─attribute_name─────┬─))──────────────────────────────────▶◀
                       └─__attribute_name__─┘
```

Type Attribute aligned

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

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

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

the placement of the type attribute can be confusing.

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

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

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

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

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

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

Type Attribute packed

Specifying the `packed` type attribute on a struct, class, union, or enumeration type indicates that the minimum amount of required memory is to be used for that type. Placement of type attribute `packed` is the same as for type attribute `aligned`, except that type attribute `packed` is silently ignored on a **typedef** declaration.

Type Attribute transparent_union

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

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

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

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

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

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

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

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

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

Restrictions

The union must be a complete union type

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

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

Examples

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

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

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

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

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

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

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

Compatible Types

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

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

Identical Types

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

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

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

Compatibility Across Separately Compiled Source Files

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

The requirements for compatibility between two structure, union, or enumerated types declared in separate source files are as follows:
- If one is declared with a tag, the other must also be declared with the same tag.
- If both are completed types, their members must correspond exactly in number, be declared with compatible types, and have matching names.

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

For structures and unions, the following additional requirements must be met for type compatibility:
- Corresponding members must be declared in the same order (applies to structures only).
- Corresponding bit fields must have the same widths.

Simple Type Specifiers

A *simple type specifier* either specifies a (previously declared) user-defined type or a *fundamental type*. A fundamental type is a one that is built into the language. The following outline shows the categories of fundamental types:
- Arithmetic types
  - Integral types
    - **char**
    - **wchar_t**
    - Signed integer types
      - **signed char**
      - **short int**
      - **int**
      - **long int**
      - **long long int**
    - Unsigned integer types
      - **_Bool**
      - **unsigned char**
      - **unsigned short int**
      - **unsigned int**
      - **unsigned long int**
      - **unsigned long long int**
  - Floating-point types
    - **float**
    - **double**
    - **long double**
- **void**

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

Boolean Variables

A Boolean variable can be used to hold the integer values 0 or 1. To declare a Boolean variable in C, use the `bool` macro, which is defined in the header file `<stdbool.h>`. A Boolean variable may not be further qualified by the specifiers `signed`, `unsigned`, `short`, or `long`.

A Boolean variable can be used to hold the integer values 0 or 1. To declare a Boolean variable, use the `bool` macro, which is defined in the header file `<stdbool.h>`. A Boolean variable may not be further qualified by the specifiers `signed`, `unsigned`, `short`, or `long`.

The Boolean type is unsigned and has the lowest ranking in its category of standard unsigned integer types. In simple assignments, if the left operand is a Boolean type, then the right operand must be either an arithmetic type or a pointer. An object declared as a Boolean type uses 1 byte of storage space, which is large enough to hold the values 0 or 1.

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

char and wchar_t Type Specifiers

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

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

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

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

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

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

Examples of the char Type Specifier

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

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

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

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

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

```
char *string_pointer;
```

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

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

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

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

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

Floating-Point Variables

There are three types of floating-point variables:
- **float**
- **double**
- **long double**

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

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

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

Examples of Floating-Point Data Types

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

```
double pi;
```

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

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

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

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

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

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

```
extern long double maximum;
```

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

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

Related References
- "Floating-Point Literals" on page 18
- "Assignment Expressions" on page 111

Integer Variables

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

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

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

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

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

Examples of Integer Data Types

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

```
short int flag;
```

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

```
int result;
```

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

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

void Type

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

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

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

Example of void Type

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

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

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

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

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

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

   return(0);
}

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

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

Compound Types

Standard C does not formally define the concept of a compound type. However, the notion of a compound type exists in C.

You are considered to be using a compound type when you construct any of the following:
- An array of objects of a given type

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

Structures

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

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

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

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

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

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

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

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

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

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

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

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

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

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

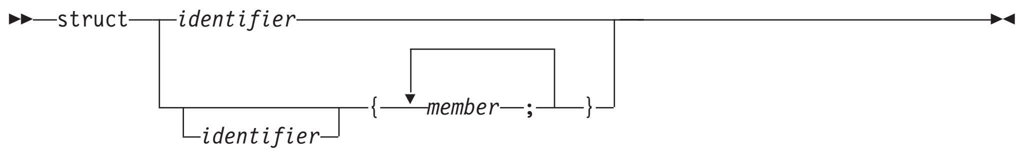
Compatible Structures

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

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

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

A declaration of a structure data type has the form:



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

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

Defining Structure Members

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

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

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

Flexible Array Members

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

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

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

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

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

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

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

```
►►──array_identifier[0]──────────────────────────────────────────────►◄
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Example

The following definition shows a completely initialized structure:

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

The values of `perm_address` are:

Member	**Value**
`perm_address.street_no`	`3`
`perm_address.street_name`	address of string `"Savona Dr."`
`perm_address.city`	address of string `"Dundas"`
`perm_address.prov`	address of string `"Ontario"`
`perm_address.postal_code`	address of string `"L4B 2A1"`

The following definition shows a partially initialized structure:

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

The values of `temp_address` are:

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

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

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

For example:

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

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

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

For example:

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

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

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

In addition to the alignment mode set by the `align` compiler option, a `#pragma options align` directive can be used to set the alignment mode. Because the `align` option in effect at the opening brace of the structure determines how the structure is mapped, a `#pragma options align` nested within a structure will only effect the definitions of structures that have the opening brace following the pragma.

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

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

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

Declaring and Using Bit Fields in Structures: 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.

The syntax for declaring a bit field is as follows:

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

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

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

- **int**
- **short**, **signed short**, **unsigned short**
- **char**, **signed char**, **unsigned char**
- **long**, **signed long**, **unsigned long**
- **long long**, **signed long long**, **unsigned long long**

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

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

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

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

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

The following restrictions apply to bit fields. You cannot:

- Define an array of bit fields
- Take the address of a bit field
- Have a pointer to a bit field
- Have a reference to a bit field

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

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

Alignment of Bit Fields

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

The following example demonstrates padding. 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

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

The following expression sets the `light` field to `1`:

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

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

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

Related References
- ″Aligning data in aggregates″ in *XL C Programming Guide*
- ″`-qalign`″ in *XL C Compiler Reference*

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

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

#include <stdio.h>

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

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

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

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

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

   return(0);
}
```

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

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

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

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

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

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

advances the pointer to the next object in the list.

Related References
- “Incomplete Types” on page 66

Unions

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

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

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

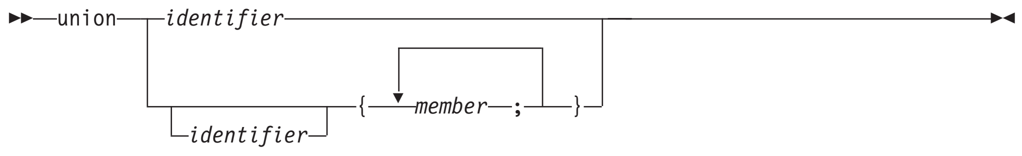
```
union { /* ... */ } caw = { .any_member = 13 };
```

Compatible Unions

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

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

A union definition has the following form:



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

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

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

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

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

For example:

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

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

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

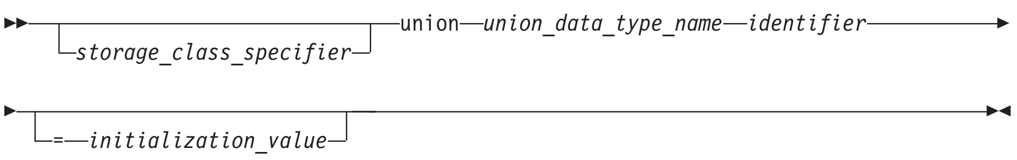
A union can represent only one of its members at a time. In the example, the union `people` contains either `age`, `birthday`, or `weight` but never more than one of these. The `printf` statement in the following example does not give the correct result because `people.age` replaces the value assigned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

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

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

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

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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

The following statement assigns a character to `input`:

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

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

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

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

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

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

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

Enumerations

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

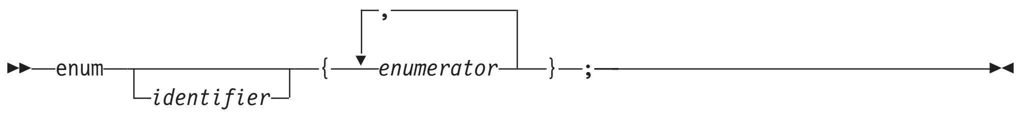
You can define an enumeration data type and all variables that have that enumeration type in one statement, or you can declare an enumeration type separately from the definition of variables of that type. The identifier associated with the data type (not an object) is called an *enumeration tag*. Each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

Compatible Enumerations

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

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

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



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

In C, each enumerator represents an integer value.

An enumerator has the form:

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

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

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

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

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

In C, enumeration constants have type **int**. Like integer constants, the type of an enumeration constant can be modified by the suffixes for unsignedness (**u** or **U**), long integer (**l** or **L**), and long long integer (**ll** or **LL**). 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>`).

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

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

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

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

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

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

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

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

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

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

>─┬────────────────────────┬───────────────────────────────────────────────────><
  └─=─enumeration_constant─┘
```

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

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

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

The type specifier `enum grain` indicates that the value of `g_food` is a member of the enumerated data type `grain`.

Defining an Enumeration Type and Enumeration Objects: You can define a type and a variable in one statement by using a declarator and an optional initializer after the type definition. To specify a storage class specifier for the variable, you must put the storage class specifier at the beginning of the declaration. For example:

```
register enum score { poor=1, average, good } rating = good;
```

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.

Example Program Using Enumerations: The following program receives an integer as input. The output is a sentence that gives the French name for the weekday that is associated with the integer. If the integer is not associated with a weekday, the program prints "`C'est le mauvais jour.`"

```
/**
 ** Example program using enumerations
 **/

#include <stdio.h>

enum days {
            Monday=1, Tuesday, Wednesday,
            Thursday, Friday, Saturday, Sunday
          } weekday;

void french(enum days);

int main(void)
{
   int num;

   printf("Enter an integer for the day of the week.  "
          "Mon=1,...,Sun=7\n");
   scanf("%d", &num);
   weekday=num;
   french(weekday);
   return(0);
}
void french(enum days weekday)
{
   switch (weekday)
   {
      case Monday:
         printf("Le jour de la semaine est lundi.\n");
         break;
      case Tuesday:
         printf("Le jour de la semaine est mardi.\n");
         break;
      case Wednesday:
         printf("Le jour de la semaine est mercredi.\n");
         break;
      case Thursday:
         printf("Le jour de la semaine est jeudi.\n");
         break;
      case Friday:
         printf("Le jour de la semaine est vendredi.\n");
         break;
      case Saturday:
         printf("Le jour de la semaine est samedi.\n");
         break;
      case Sunday:
         printf("Le jour de la semaine est dimanche.\n");
         break;
      default:
         printf("C'est le mauvais jour.\n");
   }
}
```

Complex Types

Complex types consist of two parts: a real part and an imaginary part. Imaginary types consist of only the imaginary part.

There are three type specifiers for complex types:
- **float**
- **double**
- **long double**

To declare a data object that is a complex type, use the one of the following type specifiers:

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

The imaginary unit I is a constant of type **float complex**. The predefined macro `_Complex_I` represents a constant expression of type `const float _Complex`, with the value of the imaginary unit.

The complex type and the real floating type are collectively called the *floating types*. Each floating type has a corresponding real type. For a real floating type, it is the same type. For a complex type, it is the type given by deleting the keyword **_Complex** from the type name.

The representation and alignment requirements of a complex type are the same as an array type containing two elements of the corresponding real type. The real part is equal to the first element; the imaginary part is equal to the second element.

Arithmetic conversions are the same as those for the real type of the complex type. If either operand is a complex type, the result is a complex type, and the operand having the smaller type for its real part is promoted to the complex type corresponding to the larger of the real types. For example, a **double _Complex** added to a **float _Complex** will yield a result of type **double _Complex**.

When casting a complex type to a real type, the imaginary part is dropped. When the value of a real type is converted to a complex type, the real part of the complex result value is determined by the rules of conversion to the corresponding real type, and the imaginary part of the complex result value is a positive zero or an unsigned zero.

The equality and inequality operators have the same behavior as for real types. None of the relational operators may have a complex type as an operand.

The C99 complex types are defined in the header file `/usr/include/complex.h`.

Related References
- "Complex Literals" on page 20

Type Qualifiers

C recognizes three type qualifiers, **const**, **volatile**, and **restrict**. The type qualifier **restrict** may only be applied to pointers.

Syntax for the const and volatile keywords

For a **volatile** or **const** pointer, you must put the keyword between the * and the identifier. For example:

```
int * volatile x;        /* x is a volatile pointer to an int */
int * const y = &z;      /* y is a const pointer to the int variable z */
```

For a pointer to a **volatile** or **const** data object, the type specifier, qualifier, and storage class specifier can be in any order. For example:

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

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

In the following example, the pointer to `y` is a constant. You can change the value that `y` points to, but you cannot change the value of `y`:

```
int * const y
```

In the following example, the value that `y` points to is a constant integer and cannot be changed. However, you can change the value of `y`:

```
const int * y
```

For other types of **volatile** and **const** variables, the position of the keyword within the definition (or declaration) is less important. For example:

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

provides the same storage as:

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

In both examples, only the structure variable `group` receives the **volatile** qualifier. Similarly, if you specified the **const** keyword instead of **volatile**, only the structure variable `group` receives the **const** qualifier. The **const** and **volatile** qualifiers when applied to a structure, union, or class also apply to the members of the structure, union, or class.

Although enumeration, class, structure, and union variables can receive the **volatile** or **const** qualifier, enumeration, class, structure, and union tags do not carry the **volatile** or **const** qualifier. For example, the `blue` structure does not carry the **volatile** qualifier:

```
volatile struct whale {
                      int weight;
                      char name[8];
                 } beluga;
struct whale blue;
```

The keywords **volatile** and **const** cannot separate the keywords **enum**, **class**, **struct**, and **union** from their tags.

You can declare or define a **volatile** or **const** function only if it is a nonstatic member function. You can define or declare any function to return a pointer to a **volatile** or **const** function.

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

You can put more than one qualifier on a declaration: the compiler ignores duplicate type qualifiers.

The const Type Qualifier

The **const** qualifier explicitly declares a data object as something that cannot be changed. Its value is set at initialization. You cannot use **const** data objects in expressions requiring a modifiable lvalue. For example, a **const** data object cannot appear on the lefthand side of an assignment statement.

An object that is declared **const** is guaranteed to remain constant for its lifetime, not throughout the entire execution of the program. For this reason, a `const` object cannot be used in constant expressions. In the following example, the `const` object `k` is declared within `foo`, is initialized to the value of `foo`'s argument, and remains constant until the function returns. In C, `k` cannot be used to specify the length of an array because that value will not be known until `foo` is called.

```
void foo(int j)
{
   const int k = j;
   int ary[k];     /* Violates rule that the length of each
                      array must be known to the compiler   */
}
```

In C, a `const` object that is declared outside a block has external linkage and can be shared among files. In the following example, you cannot use `k` to specify the length of the array because it is probably defined in another file.

```
extern const int k;
int ary[k];     /* Another violation of the rule that the length of
                   each array must be known to the compiler   */
```

A top-level declaration of a `const` object without an explicit storage class is considered to be **extern**.

The volatile Type Qualifier

The **volatile** qualifier maintains consistency of memory access to data objects. Volatile objects are read from memory each time their value is needed, and written back to memory each time they are changed. The **volatile** qualifier declares a data object that can have its value changed in ways outside the control or detection of the compiler (such as a variable updated by the system clock). The compiler is thereby notified not to apply certain optimizations to code referring to the object.

Accessing any lvalue expression that is **volatile**-qualified produces a side effect. A side effect means that the state of the execution environment changes.

References to an object of type ″pointer to **volatile**″ may be optimized, but no optimization can occur to references to the object to which it points. An explicit cast must be used to assign a value of type ″pointer to **volatile** `T`″ to an object of type ″pointer to `T`″. The following shows valid uses of **volatile** objects.

```
volatile int * pvol;
int *ptr;
pvol = ptr;              /* Legal */
ptr = (int *)pvol;       /* Explicit cast required */
```

A signal-handling function may store a value in a variable of type `sig_atomic_t`, provided that the variable is declared **volatile**. This is an exception to the rule that a signal-handling function may not access variables with static storage duration.

The restrict Type Qualifier

The **restrict** type qualifier may only be applied to a pointer. A pointer declaration that uses this type qualifier establishes a special association between the pointer and the object it accesses, making that pointer and expressions based on that pointer, the only ways to directly or indirectly access the value of that object.

A pointer is the address of a location in memory. More than one pointer can access the same chunk of memory and modify it during the course of a program. The **restrict** type qualifier is an indication to the compiler that, if the memory addressed by the **restrict**-qualified pointer is modified, no other pointer will access that same memory. The compiler may choose to optimize code involving **restrict**-qualified pointers in a way that might otherwise result in incorrect behavior. It is the responsibility of the programmer to ensure that **restrict**-qualified pointers are used as they were intended to be used. Otherwise, undefined behavior may result.

If a particular chunk of memory is not modified, it can be aliased through more than one restricted pointer.

The following example shows restricted pointers as parameters of `foo()`, and how an unmodified object can be aliased through two restricted pointers.

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

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

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

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

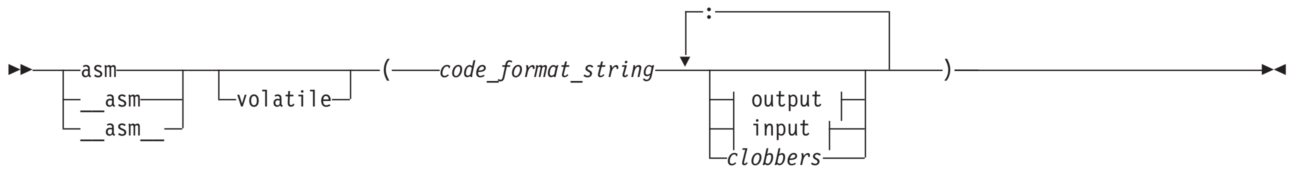
Related References
- "Type Qualifiers" on page 61

The asm Declaration

The keyword **asm** stands for assembly code. When compiled under strict language levels, the compiler recognizes and ignores the keyword **asm** in a declaration.

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

The syntax is as follows:



input:

```
   ┌─,──────────────────────────────┐
   ▼                                │
├───constraint──(──C_expression──)──┴──────────────────────────────────────────────┤
```

output:

```
   ┌─,──────────────────────────────┐
   ▼                                │
├───constraint──(──C_expression──)──┴──────────────────────────────────────────────┤
```

where

volatile
Instructs the compiler that the assembler instructions may update memory not listed in *output*, *input*, or *clobbers*.

code_format_string
Is the source text of the `asm` instructions and is a string literal similar to a `printf` format specifier.

input Is a comma-separated list of input operands.

output Is a comma-separated list of output operands.

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

constraint
Is a string literal specifying the constraints for the operand, one character per constraint.

C_expression
Is a C or C++ expression whose value is used as the operand for the `asm` instruction. Output operands must be modifiable lvalues.

The following constraints are supported.

=	Write-only operand.
+	Read and write operand.
&	An operand may be modified before the instruction is finished using the input operands; a register that is used as input should not be reused here.
b	Use a general register other than zero.
f	Use a floating-point register.
g	Use a general register, memory, or immediate operand.
i	An immediate integer operand.
m	A memory operand supported by the machine.
n	Handle in the same way as *i*.
o	Handle in the same way as *m*.
r	Use a general register.
v	Use a vector register.
0, 1, 2, ...8	A matching constraint. Allocate the same register in output as in the corresponding input.
I, J, K, M, N, O, P, G, S, T	Constant values. Fold the expression in the operand and substitute the value into the % specifier.

Restrictions

The number of instructions in an **asm** statement is limited to a maximum of 63.

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.

Incomplete Types

The following are incomplete types:
- Type **void**
- Array of unknown size
- Arrays of elements that are of incomplete type
- Structure, union, or enumerations that have no definition

void is an incomplete type that cannot be completed. Incomplete structure or union and enumeration tags must be completed before being used to declare an object, although you can define a pointer to an incomplete structure or union.

An array with an unspecified size is an incomplete type. However, if, instead of a constant expression, the array size is specified by [*], indicating a variable length array, the size is considered as having been specified, and the array type is then considered a complete type.

If the function declarator is not part of a definition of that function, parameters may have incomplete type. The parameters may also have variable length array type, indicated by the [*] notation.

The following examples illustrate incomplete types:

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

void is an incomplete type that cannot be completed. Incomplete structure, union, or enumeration tags must be completed before being used to declare an object. However, you can define a pointer to an incomplete structure or union.

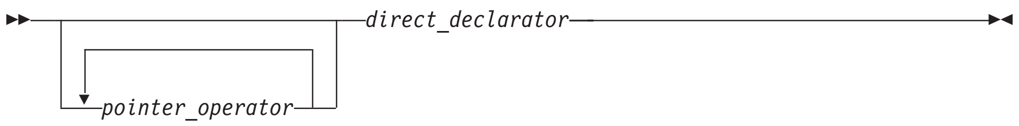
Chapter 4. Declarators

A *declarator* designates a data object or function. Declarators appear in most data definitions and declarations and in some type definitions.

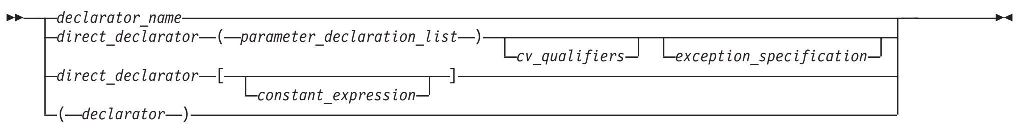
In a declarator, you can specify the type of an object to be an array, a pointer, or a reference. You can also perform initialization in a declarator.

A declarator has the form:

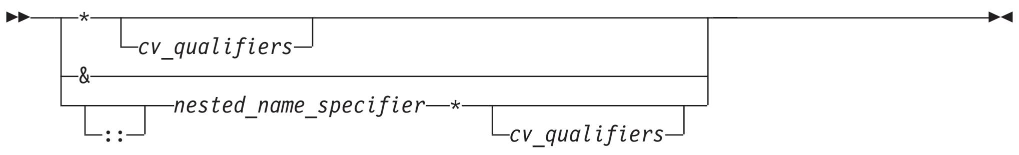
declarator



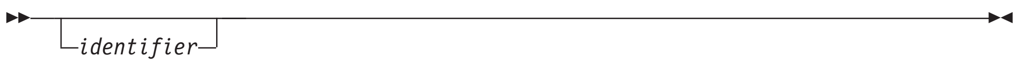
direct_declarator



pointer_operator



declarator_id



Notes on the declarator syntax

- The *cv_qualifiers* variable represents one or a combination of **const** and **volatile**. In C, you cannot declare or define a **volatile** or **const** function.

The following table provides some examples of declarators:

Example	Description
`int owner`	`owner` is an **int** data object.
`int *node`	`node` is a pointer to an **int** data object.
`int names[126]`	`names` is an array of 126 **int** elements.
`int *action( )`	`action` is a function returning a pointer to an **int**.
`volatile int min`	`min` is an **int** that has the **volatile** qualifier.
`int * volatile volume`	`volume` is a **volatile** pointer to an **int**.

Example	Description
`volatile int * next`	`next` is a pointer to a **volatile int**.
`volatile int * sequence[5]`	`sequence` is an array of five pointers to **volatile int** objects.
`extern const volatile int clock`	`clock` is a constant and volatile integer with static storage duration and external linkage.

Initializers

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

The initialization properties and special requirements of each data type are described in the section for that data type.

The initializer consists of the `=` symbol followed by an initial *expression* or a brace-enclosed list of initial expressions separated by commas. Individual expressions must be separated by commas, and groups of expressions can be enclosed in braces and separated by commas. Braces (`{ }`) are optional if the initializer for a character string is a string literal. The number of initializers must not be greater than the number of elements to be initialized. The initial expression evaluates to the first value of the data object.

To assign a value to an arithmetic or pointer type, use the simple initializer: `=` *expression*. For example, the following data definition uses the initializer `= 3` to set the initial value of `group` to `3`:

```
int group = 3;
```

For unions, structures, and aggregate classes (classes with no constructors, base classes, virtual functions, or private or protected members), the set of initial expressions must be enclosed in braces unless the initializer is a string literal.

In an array, structure, or union initialized using a brace-enclosed initializer list, any members or subscripts that are not initialized are implicitly initialized to zero of the appropriate type.

Example

In the following example, only the first eight elements of the array `grid` are explicitly initialized. The remaining four elements that are not explicitly initialized are initialized as if they were explicitly initialized to zero.

```
static short grid[3] [4] = {0, 0, 0, 1, 0, 0, 1, 1};
```

The initial values of `grid` are:

Element	Value	Element	Value
`grid[0] [0]`	`0`	`grid[1] [2]`	`1`
`grid[0] [1]`	`0`	`grid[1] [3]`	`1`
`grid[0] [2]`	`0`	`grid[2] [0]`	`0`
`grid[0] [3]`	`1`	`grid[2] [1]`	`0`
`grid[1] [0]`	`0`	`grid[2] [2]`	`0`
`grid[1] [1]`	`0`	`grid[2] [3]`	`0`

Pointers

A *pointer* type variable holds the address of a data object or a function. A pointer can refer to an object of any one data type; it cannot refer to a bit field or a reference. A pointer is classified as a scalar type, which means that it can hold only one value at a time.

Some common uses for pointers are:
- To access dynamic data structures such as linked lists, trees, and queues.
- To access elements of an array or members of a structure or C++ class.
- To access an array of characters as a string.
- To pass the address of a variable to a function. (In C++, you can also use a reference to do this.) By referencing a variable through its address, a function can change the contents of that variable.

You cannot use pointers to reference objects that are declared with the **register** storage class specifier.

Two pointer types with the same type qualifiers are compatible if they point to objects of compatible types. The composite type for two compatible pointer types is the similarly qualified pointer to the composite type.

Declaring Pointers

The following example declares `pcoat` as a pointer to an object having type **long**:

```
long *pcoat;
```

If the keyword **volatile** appears before the `*`, the declarator describes a pointer to a **volatile** object. If the keyword **volatile** appears between the `*` and the identifier, the declarator describes a **volatile** pointer. The keyword **const** operates in the same manner as the **volatile** keyword. In the following example, `pvolt` is a constant pointer to an object having type **short**:

```
extern short * const pvolt;
```

The following example declares `pnut` as a pointer to an **int** object having the **volatile** qualifier:

```
extern int volatile *pnut;
```

The following example defines `psoup` as a **volatile** pointer to an object having type **float**:

```
float * volatile psoup;
```

The following example defines `pfowl` as a pointer to an enumeration object of type `bird`:

```
enum bird *pfowl;
```

The next example declares `pvish` as a pointer to a function that takes no parameters and returns a **char** object:

```
char (*pvish)(void);
```

Assigning Pointers

When you use pointers in an assignment operation, you must ensure that the types of the pointers in the operation are compatible.

The following example shows compatible declarations for the assignment operation:

```
float subtotal;
float * sub_ptr;
/* ... */
sub_ptr = &subtotal;
printf("The subtotal is %f\n", *sub_ptr);
```

The next example shows incompatible declarations for the assignment operation:

```
double league;
int * minor;
/* ... */
minor = &league;     /* error */
```

Initializing Pointers

The initializer is an = (equal sign) followed by the expression that represents the address that the pointer is to contain. The following example defines the variables `time` and `speed` as having type **double** and `amount` as having type pointer to a **double**. The pointer `amount` is initialized to point to `total`:

```
double total, speed, *amount = &total;
```

The compiler converts an unsubscripted array name to a pointer to the first element in the array. You can assign the address of the first element of an array to a pointer by specifying the name of the array. The following two sets of definitions are equivalent. Both define the pointer `student` and initialize `student` to the address of the first element in `section`:

```
int section[80];
int *student = section;
```

is equivalent to:

```
int section[80];
int *student = &section[0];
```

You can assign the address of the first character in a string constant to a pointer by specifying the string constant in the initializer.

The following example defines the pointer variable `string` and the string constant `"abcd"`. The pointer `string` is initialized to point to the character `a` in the string `"abcd"`.

```
char *string = "abcd";
```

The following example defines `weekdays` as an array of pointers to string constants. Each element points to a different string. The pointer `weekdays[2]`, for example, points to the string `"Tuesday"`.

```
static char *weekdays[ ] =
      {
        "Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"
      };
```

A pointer can also be initialized to null using any integer constant expression that evaluates to `0`, for example `char * a=0;`. Such a pointer is a *null pointer*. It does not point to any object.

Using Pointers

Two operators are commonly used in working with pointers, the address (&) operator and the indirection (*) operator. You can use the & operator to refer to the address of an object. For example, the assignment in the following function assigns the address of x to the variable p_to_int. The variable p_to_int has been defined as a pointer:

```
void f(int x, int *p_to_int)
{
  p_to_int = &x;
}
```

The * (indirection) operator lets you access the value of the object a pointer refers to. The assignment in the following example assigns to y the value of the object that p_to_float points to:

```
void g(float y, float *p_to_float) {
  y = *p_to_float;
}
```

The assignment in the following example assigns the value of z to the variable that *p_to_char references:

```
void h(char z, char *p_to_char) {
  *p_to_char = z;
}
```

Pointer Arithmetic

You can perform a limited number of arithmetic operations on pointers. These operations are:
- Increment and decrement
- Addition and subtraction
- Comparison
- Assignment

The increment (++) operator increases the value of a pointer by the size of the data object the pointer refers to. For example, if the pointer refers to the second element in an array, the ++ makes the pointer refer to the third element in the array.

The decrement (--) operator decreases the value of a pointer by the size of the data object the pointer refers to. For example, if the pointer refers to the second element in an array, the -- makes the pointer refer to the first element in the array.

You can add an integer to a pointer but you cannot add a pointer to a pointer.

If the pointer p points to the first element in an array, the following expression causes the pointer to point to the third element in the same array:

```
p = p + 2;
```

If you have two pointers that point to the same array, you can subtract one pointer from the other. This operation yields the number of elements in the array that separate the two addresses that the pointers refer to.

You can compare two pointers with the following operators: ==, !=, <, >, <=, and >=.

Pointer comparisons are defined only when the pointers point to elements of the same array. Pointer comparisons using the `==` and `!=` operators can be performed even when the pointers point to elements of different arrays.

You can assign to a pointer the address of a data object, the value of another compatible pointer or the `NULL` pointer.

Example Program Using Pointers

The following program contains pointer arrays:

```
/********************************************************************
**   Program to search for the first occurrence of a specified     **
**   character string in an array of character strings.            **
********************************************************************/

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

#define  SIZE  20

int main(void)
{
   static char *names[ ] = { "Jim", "Amy", "Mark", "Sue", NULL };
   char * find_name(char **, char *);
   char new_name[SIZE], *name_pointer;

   printf("Enter name to be searched.\n");
   scanf("%s", new_name);
   name_pointer = find_name(names, new_name);
   printf("name %s%sfound\n", new_name,
          (name_pointer == NULL) ? " not " : " ");
} /* End of main */



/********************************************************************
**     Function find_name.  This function searches an array of     **
**     names to see if a given name already exists in the array.   **
**     It returns a pointer to the name or NULL if the name is     **
**     not found.                                                  **
**                                                                 **
** char **arry is a pointer to arrays of pointers (existing names) **
** char *strng is a pointer to character array entered (new name)  **
********************************************************************/

char * find_name(char **arry, char *strng)
{
   for (; *arry != NULL; arry++)        /* for each name          */
   {
      if (strcmp(*arry, strng) == 0)    /* if strings match       */
         return(*arry);                 /* found it!              */
   }
   return(*arry);                       /* return the pointer     */
} /* End of find_name */
```

Interaction with this program could produce the following sessions:

Output	`Enter name to be searched.`
Input	`Mark`
Output	`name Mark found`

or:

Output	`Enter name to be searched.`
Input	`Deborah`
Output	`name Deborah not found`

Arrays

An *array* is a collection of objects of the same data type. Individual objects in an array, called *elements*, are accessed by their position in the array. The subscripting operator ([]) provides the mechanics for creating an index to array elements. This form of access is called *indexing* or *subscripting*. An array facilitates the coding of repetitive tasks by allowing the statements executed on each element to be put into a loop that iterates through each element in the array.

The C and C++ languages provide limited built-in support for an array type: reading and writing individual elements. Assignment of one array to another, the comparison of two arrays for equality, returning self-knowledge of size are operations unsupported by either language.

An array type describes contiguously allocated memory for a set of objects of a particular type. The array type is derived from the type of its elements, in what is called *array type derivation*. If array objects are of incomplete type, the array type is also considered incomplete.

Array elements may not be of type **void** or of function type. However, arrays of pointers to functions are allowed. In C++, array elements may not be of reference type or of an abstract class type.

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

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

are not compatible. The composite type of two compatible array types is an array with the composite element type. The sizes of both original types must be equivalent if they are known. If the size of only one of the original array types is known, then the composite type has that size. For example, suppose:

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

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

Except in certain contexts, an unsubscripted array name (for example, `region` instead of `region[4]`) represents a pointer whose value is the address of the first element of the array, provided that the array has previously been declared. The exceptions are when the array name passes the array itself. For example, the array name passes the entire array when it is the operand of the **sizeof** operator or the address (**&**) operator.

Similarly, an array type in the parameter list of a function is converted to the corresponding pointer type. Information about the size of the argument array is lost when the array is accessed from within the function body.

To preserve this information, which is useful for optimization, you may declare the index of the argument array using the **static** keyword. The constant expression specifies the minimum pointer size that can be used as an assumption for optimizations.

This particular usage of the **static** keyword is highly prescribed. The keyword may only appear in the outermost array type derivation and only in function parameter declarations. If the caller of the function does not abide by these restrictions, the behavior is undefined.

This language feature is available at the C99 language level.

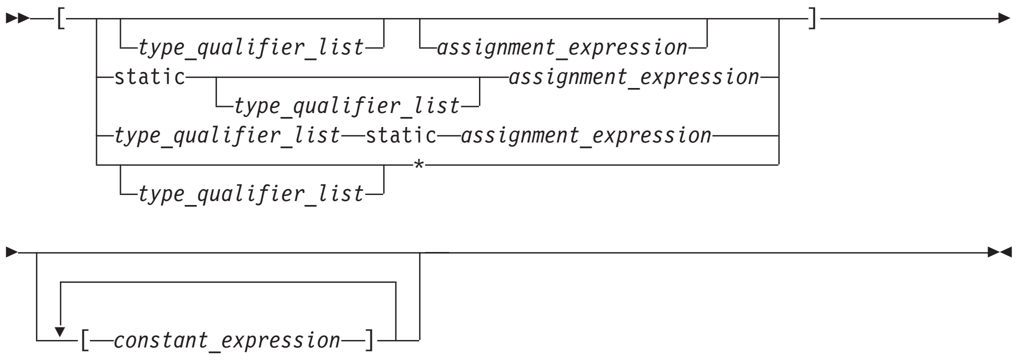
The following examples show how the feature might be used.

```
void foo(int arr [static 10]);       /* arr points to the first of at least
                                          10 ints                          */
void foo(int arr [const 10]);        /* arr is a const pointer             */
void foo(int arr [static const i]);  /* arr points to at least i ints;
                                          i is computed at run time.       */
void foo(int arr [const static i]);  /* alternate syntax to previous example */
void foo(int arr [const]);           /* const pointer to int               */
```

Declaring Arrays

The array declarator contains an identifier followed by an optional *subscript declarator*. An identifier preceded by an asterisk (*) is an array of pointers.

A subscript declarator has the form:



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

If the declaration appears in block or function scope, a nonconstant expression can be specified for the array subscript declarator, and the array is considered a variably modified type. An asterisk within the brackets of the array subscripting operator indicates a variable length array of unspecified size. In this case, the array is considered a variably modified type that can only be used in functions declarations that are not definitions (that is, in declarations with function prototype scope).

The subscript declarator describes the number of dimensions in the array and the number of elements in each dimension. Each bracketed expression, or subscript, describes a different dimension and must be a constant expression.

The following example defines a one-dimensional array that contains four elements having type **char**:

```
char
list[4];
```

The first subscript of each dimension is `0`. The array `list` contains the elements:

```
list[0]
list[1]
list[2]
list[3]
```

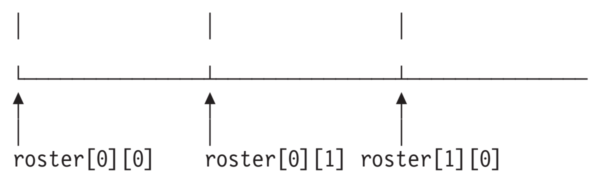
The following example defines a two-dimensional array that contains six elements of type **int**:

```
int roster[3][2];
```

Multidimensional arrays are stored in row-major order. When elements are referred to in order of increasing storage location, the last subscript varies the fastest. For example, the elements of array `roster` are stored in the order:

```
roster[0][0]
roster[0][1]
roster[1][0]
roster[1][1]
roster[2][0]
roster[2][1]
```

In storage, the elements of `roster` would be stored as:



You can leave the first (and only the first) set of subscript brackets empty in
- Array definitions that contain initializations
- **extern** declarations
- Parameter declarations

In array definitions that leave the first set of subscript brackets empty, the initializer determines the number of elements in the first dimension. In a one-dimensional array, the number of initialized elements becomes the total number of elements. In a multidimensional array, the initializer is compared to the subscript declarator to determine the number of elements in the first dimension.

Variable Length Arrays

A variable length array is an array of automatic storage duration whose length is determined at run time. The variable length array type provides a construct for allocating the right amount of storage, which can only be determined when the application is actually run.

A variable length array can be written as:

```
►►—array_identifier—[——expression——————————————]———————————————►◄
                      └——————————————————————*—┘
                        └type-qualifier-list—┘
```

If the size of the array is indicated by `*` instead of an expression, the variable length array is considered to be of unspecified size. Such arrays are considered complete types, but can only be used in declarations of function prototype scope.

A variable length array and a pointer to a variable length array are considered *variably modified types*. Declarations of variably modified types must be at either block scope or function prototype scope. Array objects declared with the `extern` storage class specifier cannot be of variable length array type. Array objects declared with the `static` storage class specifier can be a pointer to a variable length array, but not an actual variable length array. The identifiers declared with a variably modified type must be ordinary identifiers and therefore cannot be members of structures or unions. A variable length array cannot be initialized.

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

A variable length array can be used in a `typedef` expression. The `typedef` name will have only block scope. The length of the array is fixed when the `typedef` name is defined, not each time it is used.

A function parameter can be a variable length array. The necessary size expressions must be provided in the function definition. The compiler evaluates the size expression of a variably modified parameter on entry to the function. For a function declared with a variable length array as a parameter, as in the following,

```
void f(int x, int a[][x]);
```

the size of the variable length array argument must match that of the function definition.

Related References
- "Calling Functions and Passing Arguments" on page 133

Initializing Arrays

The initializer for an array is a comma-separated list of constant expressions enclosed in braces (`{ }`). The initializer is preceded by an equal sign (`=`). You do not need to initialize all elements in an array. If an array is partially initialized, elements that are not initialized receive the value `0` of the appropriate type. The same applies to elements of arrays with static storage duration. (All file-scope variables and function-scope variables declared with the **static** keyword have static storage duration.)

The following definition shows a completely initialized one-dimensional array:

```
static int number[3] = { 5, 7, 2 };
```

The array `number` contains the following values: `number[0]` is `5`, `number[1]` is `7`; `number[2]` is `2`. When you have an expression in the subscript declarator defining the number of elements (in this case `3`), you cannot have more initializers than the number of elements in the array.

The following definition shows a partially initialized one-dimensional array:

```
static int number1[3] = { 5, 7 };
```

The values of `number1` are:`number1[0]` and `number1[1]` are the same as in the previous definition, but `number1[2]` is `0`.

Instead of an expression in the subscript declarator defining the number of elements, the following one-dimensional array definition defines one element for each initializer specified:

```
static int item[ ] = { 1, 2, 3, 4, 5 };
```

The compiler gives `item` the five initialized elements, because no size was specified and there are five initializers.

You can initialize a one-dimensional character array by specifying:
- A brace-enclosed comma-separated list of constants, each of which can be contained in a character
- A string constant (Braces surrounding the constant are optional)

Initializing a string constant places the null character (`\0`) at the end of the string if there is room or if the array dimensions are not specified.

The following definitions show character array initializations:

```
   static char name1[ ] = { 'J', 'a', 'n' };
   static char name2[ ] = { "Jan" };
   static char name3[4] = "Jan";
```

These definitions create the following elements:

Element	Value	Element	Value	Element	Value
name1[0]	J	name2[0]	J	name3[0]	J
name1[1]	a	name2[1]	a	name3[1]	a
name1[2]	n	name2[2]	n	name3[2]	n
		name2[3]	\0	name3[3]	\0

Note that the following definition would result in the null character being lost:

```
   static char name3[3]="Jan";
```

You can initialize a multidimensional array using any of the following techniques:
- Listing the values of all elements you want to initialize, in the order that the compiler assigns the values. The compiler assigns values by increasing the subscript of the last dimension fastest. This form of a multidimensional array initialization looks like a one-dimensional array initialization. The following definition completely initializes the array `month_days`:

  ```
  static month_days[2][12] =
  {
   31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31,
   31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
  };
  ```
- Using braces to group the values of the elements you want initialized. You can put braces around each element, or around any nesting level of elements. The following definition contains two elements in the first dimension (you can consider these elements as rows). The initialization contains braces around each of these two elements:

  ```
  static int month_days[2][12] =
  {
   { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
   { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
  };
  ```

- Using nested braces to initialize dimensions and elements in a dimension selectively.

The following definition explicitly initializes six elements in a 12-element array:

```
static int matrix[3][4] =
   {
     {1, 2},
     {3, 4},
     {5, 6}
   };
```

The initial values of `matrix` are shown in the following table. All other elements are initialized to zero.

Element	Value	Element	Value
`matrix[0][0]`	1	`matrix[1][2]`	0
`matrix[0][1]`	2	`matrix[1][3]`	0
`matrix[0][2]`	0	`matrix[2][0]`	5
`matrix[0][3]`	0	`matrix[2][1]`	6
`matrix[1][0]`	3	`matrix[2][2]`	0
`matrix[1][1]`	4	`matrix[2][3]`	0

Initializing Arrays Using Designated Initializers

C supports designated initializers for aggregate types. A *designator* points out a particular array element to be initialized, and is of the form "[*index*]", where *index* is a constant expression. A *designator list* is a combination of one or more designators for any of the aggregate types. A designator list followed by an equal sign constitutes a *designation*.

In the absence of designations, initialization of an array occurs in the order indicated by the initializer. When a designation appears in an initializer, the array element indicated by the designator is initialized, and subsequent initializations proceed forward in initializer-list order, overriding any previously initialized array element, and initializing to zero any array elements that are not explicitly initialized.

The declaration syntax without a designated initializer uses braces to indicate initializer lists, but is referred to as a *bracketed form*. The fully bracketed and minimally bracketed forms of initialization are less likely to be misunderstood. The following are valid declarations of the multidimensional array `matrix` that achieve the same thing. All array elements that are not explicitly initialized, such as the entire row beginning with `matrix[3][0][0]`, are initialized to zero.

```
/* minimally bracketed form */
int matrix[4][3][2] = {
    1, 0, 0, 0, 0, 0,
    2, 3, 0, 0, 0, 0,
    4, 5, 6
};

/* fully bracketed form */
int matrix[4] [3] [2] = {
    {
        { 1 },
    },
    {
        { 2, 3 },
    },
    {
        { 4, 5 },
        { 6 }
    }
};

/* incompletely but consistently bracketed initialization */
int matrix[4] [3] [2] = {
    { 1 },
    { 2, 3 },
    { 4, 5, 6 }
};
```

The overriding of previous subobject initializations during an array initialization is necessary behavior for the designated initializer. To illustrate this, a single designator is used to ″allocate″ space from both ends of an array. The designated initializer, [MAX-5] = 8, means that the array element at subscript MAX-5 should be initialized to the value 8. The array subscripting brackets must enclose a constant expression.

```
int a[MAX] = {
      1, 3, 5, 7, 9, [MAX-5] = 8, 6, 4, 2, 0
};
```

If MAX is 15, a[5] through a[9] will be initialized to zero. If MAX is 7, a[2] through a[4] will first have the values 5, 7, and 9, respectively, which are overridden by the values 8, 6, and 4. In other words, if MAX is 7, the initialization would be the same as if the declaration had been written:

```
int a[MAX] = {
      1, 3, 8, 6, 4, 2, 0
};
```

Example Programs Using Arrays

The following program defines a floating-point array called prices.

The first for statement prints the values of the elements of prices. The second for statement adds five percent to the value of each element of prices, and assigns the result to total, and prints the value of total.

```
/**
 ** Example of one-dimensional arrays
 **/

#include <stdio.h>
#define  ARR_SIZE  5

int main(void)
{
  static float const prices[ARR_SIZE] = { 1.41, 1.50, 3.75, 5.00, .86 };
  auto float total;
  int i;

  for (i = 0; i < ARR_SIZE; i++)
  {
    printf("price = $%.2f\n", prices[i]);
  }

  printf("\n");

  for (i = 0; i < ARR_SIZE; i++)
  {
    total = prices[i] * 1.05;
    printf("total = $%.2f\n", total);
  }

  return(0);
}
```

This program produces the following output:

```
price = $1.41
price = $1.50
price = $3.75
price = $5.00
price = $0.86

total = $1.48
total = $1.57
total = $3.94
total = $5.25
total = $0.90
```

The following program defines the multidimensional array `salary_tbl`. A **for** loop prints the values of `salary_tbl`.

```
/**
 ** Example of a multidimensional array
 **/

#include <stdio.h>
#define  ROW_SIZE     3
#define  COLUMN_SIZE  5

int main(void)
{
  static int
  salary_tbl[ROW_SIZE][COLUMN_SIZE] =
  {
    {  500,  550,  600,  650,  700   },
    {  600,  670,  740,  810,  880   },
    {  740,  840,  940, 1040, 1140   }
  };
  int grade , step;

  for (grade = 0; grade < ROW_SIZE; ++grade)
   for (step = 0; step < COLUMN_SIZE; ++step)
   {
     printf("salary_tbl[%d] [%d] = %d\n",
            grade, step, salary_tbl[grade] [step]);
   }

   return(0);
}
```

This program produces the following output:

```
salary_tbl[0] [0] = 500
salary_tbl[0] [1] = 550
salary_tbl[0] [2] = 600
salary_tbl[0] [3] = 650
salary_tbl[0] [4] = 700
salary_tbl[1] [0] = 600
salary_tbl[1] [1] = 670
salary_tbl[1] [2] = 740
salary_tbl[1] [3] = 810
salary_tbl[1] [4] = 880
salary_tbl[2] [0] = 740
salary_tbl[2] [1] = 840
salary_tbl[2] [2] = 940
salary_tbl[2] [3] = 1040
salary_tbl[2] [4] = 1140
```

Function Specifiers

The function specifier **inline** is used to make a suggestion to the compiler to incorporate the code of a function into the code at the point of the call. Instead of creating a single set of the function instructions in memory, the compiler is supposed to copy the code from the inline function directly into the calling function. However, a standards-compliant compiler may ignore this suggestion for better optimization.

Chapter 5. Expressions and Operators

Expressions are sequences of operators, operands, and punctuators that specify a computation. The evaluation of expressions is based on the operators that the expressions contain and the context in which they are used. An expression can result in a value and can produce side effects. A *side effect* is a change in the state of the execution environment.

ISO C heeds points in the execution sequence at which all side effects of previous evaluations are complete and no side effects of subsequent evaluations will have occurred. Such times are called *sequence points*. A scalar object may be modified only once between successive sequence points; otherwise, the result is undefined. Sequence points occur at the completion of all expressions that are not part of a larger expression, such as in the following situations:

- After the evaluation of the first operand of a logical AND &&, logical OR `||`, conditional ?:, or comma expression
- After the evaluation of the arguments in a function call
- At the end of a full declarator
- At the end of a full expression
- Before a library function returns
- After the actions of a formatted I/O function conversion specifier
- Before and after a call to a comparison function, and between any call to the comparison function and any movement of the objects passed as arguments to that function call

The term *full expression* can mean an initializer, an expression statement, the expression in a `return` statement, and the control expressions in a conditional, iterative, or `switch` statement. This includes each expression in a `for` statement.

Related References
- "Lvalues and Rvalues" on page 86

Operator Precedence and Associativity

Two operator characteristics determine how operands group with operators: *precedence* and *associativity*. Precedence is the priority for grouping different types of operators with their operands. Associativity is the left-to-right or right-to-left order for grouping operands to operators that have the same precedence. An operator's precedence is meaningful only if other operators with higher or lower precedence are present. Expressions with higher-precedence operators are evaluated first. The grouping of operands can be forced by using parentheses.

For example, in the following statements, the value of `5` is assigned to both `a` and `b` because of the right-to-left associativity of the `=` operator. The value of `c` is assigned to `b` first, and then the value of `b` is assigned to `a`.

```
b = 9;
c = 5;
a = b = c;
```

Because the order of subexpression evaluation is not specified, you can explicitly force the grouping of operands with operators by using parentheses.

In the expression

```
a + b * c / d
```

the `*` and `/` operations are performed before `+` because of precedence. `b` is multiplied by `c` before it is divided by `d` because of associativity.

The following table lists the language operators in order of precedence and shows the direction of associativity for each operator.

The comma operator has the lowest precedence. Operators that have the same rank have the same precedence.

Precedence and associativity of C 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* **--**
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*
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*
3		member selection	*object* .* *ptr_to_member*
3		member selection	*object* ->* *ptr_to_member*
4		multiplication	*expr* * *expr*
4		division	*expr* **/** *expr*
4		modulo (remainder)	*expr* **%** *expr*
5		binary addition	*expr* **+** *expr*
5		binary subtraction	*expr* **-** *expr*
6		bitwise shift left	*expr* **<<** *expr*
6		bitwise shift right	*expr* **>>** *expr*
7		less than	*expr* **<** *expr*
7		less than or equal to	*expr* **<=** *expr*
7		greater than	*expr* **>** *expr*
7		greater than or equal to	*expr* **>=** *expr*
8		equal	*expr* **==** *expr*
8		not equal	*expr* **!=** *expr*
9		bitwise AND	*expr* **&** *expr*
10		bitwise exclusive OR	*expr* **^** *expr*
11		bitwise inclusive OR	*expr* | *expr*
12		logical AND	*expr* **&&** *expr*
13		logical inclusive OR	*expr* || *expr*
14		conditional expression	*expr* **?** *expr* **:** *expr*

Precedence and associativity of C operators

Rank	Right Associative?	Operator Function	Usage
15	yes	simple assignment	*lvalue* = *expr*
15	yes	multiply and assign	*lvalue* *= *expr*
15	yes	divide and assign	*lvalue* /= *expr*
15	yes	modulo and assign	*lvalue* %= *expr*
15	yes	add and assign	*lvalue* += *expr*
15	yes	subtract and assign	*lvalue* -= *expr*
15	yes	shift left and assign	*lvalue* <<= *expr*
15	yes	shift right and assign	*lvalue* >>= *expr*
15	yes	bitwise AND and assign	*lvalue* &= *expr*
15	yes	bitwise exclusive OR and assign	*lvalue* ^= *expr*
15	yes	bitwise inclusive OR and assign	*lvalue* |= *expr*
16		comma (sequencing)	*expr* , *expr*

The order of evaluation for function call arguments or for the operands of binary operators is not specified. Avoid writing ambiguous expressions such as:

```
z = (x * ++y) / func1(y);
func2(++i, x[i]);
```

In the example above, `++y` and `func1(y)` might not be evaluated in the same order by all C language implementations. If `y` has the value of 1 before the first statement, it is not known whether or not the value of 1 or 2 is passed to `func1()`. In the second statement, if `i` has the value of 1 before the expression is evaluated, it is not known whether `x[1]` or `x[2]` is passed as the second argument to `func2()`.

Examples of Expressions and Precedence

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

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

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

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

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

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

in the following ways (as indicated by parentheses):

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

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

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

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

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

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

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

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

Lvalues and Rvalues

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

- An array type
- An incomplete type
- A **const**-qualified type
- An object is a structure or union type and one of its members has a **const**-qualified type

Because these lvalues are not modifiable, they cannot appear on the left side of an assignment statement.

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

ISO C defines a *function designator* as an expression that has function type A function designator is distinct from an object type or an lvalue. It can be the name of a function or the result of dereferencing a function pointer. The C language also differentiates between its treatment of a function pointer and an object pointer.

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

Operator	Requirement
& (unary)	Operand must be an lvalue.
++ --	Operand must be an lvalue. This applies to both prefix and postfix forms.
= += -= *= %= <<= >>= &= ^= |=	Left operand must be an lvalue.

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

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

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

The remainder of this section is platform-specific and pertains to C only.

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

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

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

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

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

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

Related References
- “Lvalue-to-Rvalue Conversions” on page 116

Primary Expressions

Primary expressions fall into the following general categories:
- Names (identifiers)
- Literals (constants)
- 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.

Primary expressions: Names

Name declared as	Evaluates to	Is an lvalue
Variable of arithmetic, pointer, enumeration, structure, or union type	An object of that type	Lvalue
Enumeration constant	The associated integer value	Not an lvalue
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.	Not an lvalue
Function	That function. In contexts subject to conversions, a pointer to that function, except where the name is used as the argument to the `sizeof` operator, or as the function in a function call expression.	Not an lvalue

As an expression, a name may not refer to a label, `typedef` name, structure component name, union component name, structure tag, union tag, or enumeration tag. Names that can be referred to by a name in an expression reside in a name space that is separate from that of names for these purposes. Some of these names may be referred to within expressions by means of special constructs. For example, the dot or arrow operators may be used to refer to structure and union component names; `typedef` names may be used in casts or as an argument to the `sizeof` operator.

Literals

A literal is a numeric constant or string literal. When a literal is evaluated as an expression, its value is a constant. A lexical constant is never an lvalue. However, a string literal is an lvalue.

Related References
- "Literals" on page 16

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

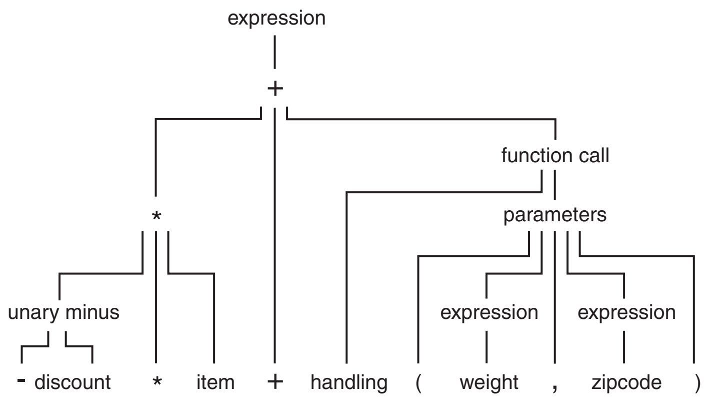
The **sizeof** operator applied to a variable length array type is evaluated at run time, and therefore is not a constant expression.

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 enum constant.
- In a bit-field width specifier.
- In the preprocessor **#if** statement. (Enumeration constants, address constants, and **sizeof** cannot be specified in a preprocessor **#if** statement.)

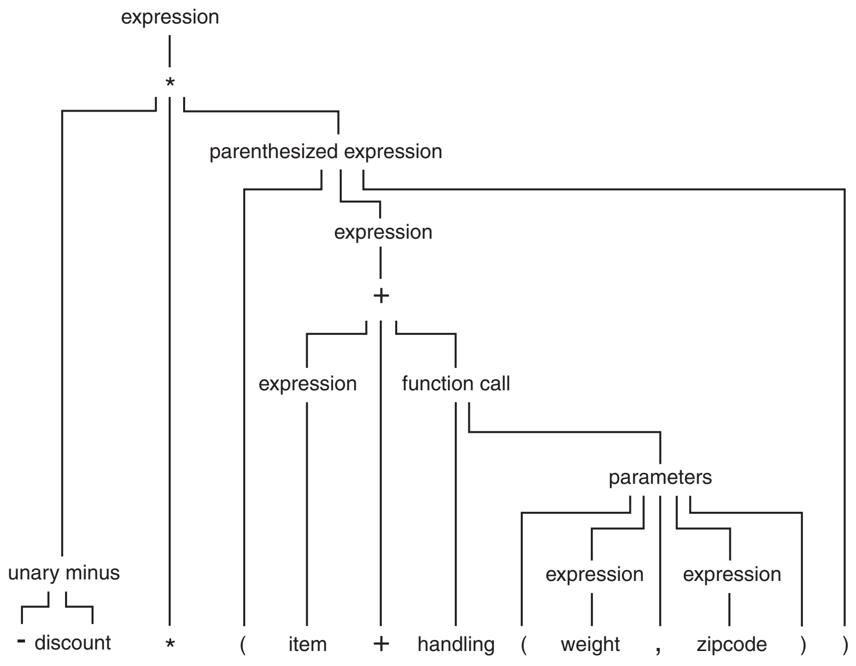
Related References
- "sizeof Operator" on page 98

Parenthesized Expressions ( )

Use parentheses to explicitly force the order of expression evaluation. The following expression does not use parentheses to group operands and operators. The parentheses surrounding `weight, zipcode` are used to form a function call. Note how the compiler groups the operands and operators in the expression according to the rules for operator precedence and associativity:



The following expression is similar to the previous expression, but it contains parentheses that change how the operands and operators are grouped:



In an expression that contains both associative and commutative operators, you can use parentheses to specify the grouping of operands with operators. The parentheses in the following expression guarantee the order of grouping operands with the operators:

```
x = f + (g + h);
```

Postfix Expressions

Postfix operators are operators that appear after their operands. A *postfix expression* is a primary expression, or a primary expression that contains a postfix operator. The following summarizes the available postfix operators:

Precedence and associativity of postfix operators

Rank	Right Associative?	Operator Function	Usage
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* **--**

Function Call Operator ( )

A *function call* is an expression containing a simple type name and a parenthesized argument list. The argument list can contain any number of expressions separated by commas. It can also be empty.

For example:

```
stub()
overdue(account, date, amount)
notify(name, date + 5)
report(error, time, date, ++num)
```

Any function may call itself except for the function `main`.

Type of a Function Call

The type of a function call expression is the return type of the function. This type can either be a complete type, a reference type, or the type **void**. A function call is an lvalue if and only if the type of the function is a reference.

Arguments and Parameters

A *function argument* is an expression that you use within the parentheses of a function call. A *function parameter* is an object or reference declared within the parentheses of a function declaration or definition. When you call a function, the arguments are evaluated, and each parameter is initialized with the value of the corresponding argument. The semantics of argument passing are identical to those of assignment.

A function can change the values of its non-const parameters, but these changes have no effect on the argument unless the parameter is a reference type.

Linkage and Function Calls

In C, if a function definition has external linkage and a return type of **int**, calls to the function can be made before it is explicitly declared because an implicit declaration of `extern int func();` is assumed.

Type Conversions of Arguments

Arguments that are arrays or functions are converted to pointers before being passed as function arguments.

Arguments passed to nonprototyped C functions undergo conversions: type **short** or **char** parameters are converted to **int**, and **float** parameters to **double**. Use a cast expression for other conversions.

The compiler compares the data types provided by the calling function with the data types that the called function expects and performs necessary type conversions. For example, when function `funct` is called, argument `f` is converted to a **double**, and argument `c` is converted to an **int**:

```
char * funct (double d, int i);
     /* ... */
int main(void)
{
   float f;
   char c;
   funct(f, c) /* f is converted to a double, c is converted to an int */
   return 0;
}
```

Evaluation Order of Arguments

The order in which arguments are evaluated is not specified. Avoid such calls as:

```
method(sample1, batch.process--, batch.process);
```

In this example, `batch.process--` might be evaluated last, causing the last two arguments to be passed with the same value.

Example of Function Calls

In the following example, `main` passes `func` two values: `5` and `7`. The function `func` receives copies of these values and accesses them by the identifiers: `a` and `b`. The function `func` changes the value of `a`. When control passes back to `main`, the actual values of `x` and `y` are not changed. The called function `func` only receives copies of the values of `x` and `y`, not the variables themselves.

```
/**
 ** This example illustrates function calls
 **/

#include <stdio.h>

void func (int a, int b)
{
   a += b;
   printf("In func, a = %d    b = %d\n", a, b);
}

int main(void)
{
   int x = 5, y = 7;
   func(x, y);
   printf("In main, x = %d    y = %d\n", x, y);
   return 0;
}
```

This program produces the following output:

```
In func, a = 12   b = 7
In main, x = 5    y = 7
```

Array Subscripting Operator [ ]

A postfix expression followed by an expression in [ ] (brackets) specifies an element of an array. The expression within the brackets is referred to as a *subscript*. The first element of an array has the subscript zero.

By definition, the expression `a[b]` is equivalent to the expression `*((a) + (b))`, and, because addition is associative, it is also equivalent to `b[a]`. Between expressions `a` and `b`, one must be a pointer to a type `T`, and the other must have integral or enumeration type. The result of an array subscript is an lvalue. The following example demonstrates this:

```
#include <stdio.h>

int main(void) {
  int a[3] = { 10, 20, 30 };
  printf("a[0] = %d\n", a[0]);
  printf("a[1] = %d\n", 1[a]);
  printf("a[2] = %d\n", *(2 + a));
  return 0;
}
```

The following is the output of the above example:

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

C99 allows array subscripting on arrays that are not lvalues. However, using the address of a non-lvalue as an array subscript is still not allowed. The following example is valid in C99, but not in C89:

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

The first element of each array has the subscript `0`. The expression `contract[35]` refers to the 36th element in the array `contract`.

In a multidimensional array, you can reference each element (in the order of increasing storage locations) by incrementing the right-most subscript most frequently.

For example, the following statement gives the value `100` to each element in the array `code[4][3][6]`:

```
for (first = 0; first < 4; ++first)
   {
   for (second = 0; second < 3; ++second)
      {
      for (third = 0; third < 6; ++third)
         {
         code[first][second][third] =
         100;
         }
      }
   }
```

Dot Operator .

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

The value of the expression is the value of the selected member. If the postfix expression and the name are lvalues, the expression value is also an lvalue. If the postfix expression is type-qualified, the same type qualifiers will apply to the designated member in the resulting expression.

Arrow Operator −>

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

The value of the expression is the value of the selected member. If the name is an lvalue, the expression value is also an lvalue. If the expression is a pointer to a qualified type, the same type-qualifiers will apply to the designated member in the resulting expression.

Related References
- "Dot Operator ." on page 93

Unary Expressions

A *unary expression* contains one operand and a unary operator. All unary operators have the same precedence and have right-to-left associativity. A unary expression is therefore a postfix expression.

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

The following table summarizes the operators for unary expressions:

Precedence and associativity of unary operators

Rank	Right Associative?	Operator Function	Usage
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	complement	**~** *expr*
2	yes	not	**!** *expr*
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*

C99 adds the unary operator `_Pragma`, which allows a preprocessor macro to contain a pragma directive.

XL C/C++ extends the C99 and C++ standards to support the unary operators `__real__` and `__imag__`. These operators provide the ability to extract the real and imaginary parts of a complex type. These extensions have been implemented to ease the porting applications developed with GNU C and C++.

Related References
- "Complex Literals" on page 20

Increment ++

The ++ (increment) operator adds 1 to the value of a scalar operand, or if the operand is a pointer, increments the operand by the size of the object to which it points. The operand receives the result of the increment operation. The operand must be a modifiable lvalue of arithmetic or pointer type.

You can put the ++ before or after the operand. If it appears before the operand, the operand is incremented. The incremented value is then used in the expression. If you put the ++ after the operand, the value of the operand is used in the expression *before* the operand is incremented. For example:

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

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

```
int temp, temp1, temp2;

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

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

The usual arithmetic conversions on the operand are performed.

Decrement −−

The `--` (decrement) operator subtracts 1 from the value of a scalar operand, or if the operand is a pointer, decreases the operand by the size of the object to which it points. The operand receives the result of the decrement operation. The operand must be a modifiable lvalue.

You can put the `--` before or after the operand. If it appears before the operand, the operand is decremented, and the decremented value is used in the expression. If the `--` appears after the operand, the current value of the operand is used in the expression and the operand is decremented.

For example:

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

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

```
int temp, temp1, temp2;

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

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

The usual arithmetic conversions are performed on the operand.

Unary Plus +

The `+` (unary plus) operator maintains the value of the operand. The operand can have any arithmetic type or pointer type. The result is not an lvalue.

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

Note: Any plus sign in front of a constant is not part of the constant.

Unary Minus −

The `-` (unary minus) operator negates the value of the operand. The operand can have any arithmetic type. The result is not an lvalue.

For example, if `quality` has the value `100`, `-quality` has the value `-100`.

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

Note: Any minus sign in front of a constant is not part of the constant.

Logical Negation !

The `!` (logical negation) operator determines whether the operand evaluates to `0` (false) or nonzero (true).

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;
```

Bitwise Negation ~

The `~` (bitwise negation) operator yields the bitwise complement of the operand. In the binary representation of the result, every bit has the opposite value of the same bit in the binary representation of the operand. The operand must have an integral type. The result has the same type as the operand but is not an lvalue.

Suppose `x` represents the decimal value `5`. The 16-bit binary representation of `x` is:

```
0000000000000101
```

The expression `~x` yields the following result (represented here as a 16-bit binary number):

```
1111111111111010
```

Note that the `~` character can be represented by the trigraph `??-`.

The 16-bit binary representation of `~0` is:

```
1111111111111111
```

Address &

The `&` (address) operator yields a pointer to its operand. The operand must be an lvalue, a function designator, or a qualified name. It cannot be a bit field, nor can it have the storage class **register**.

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

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

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

```
p_to_y = &y;
```

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

Related References
- “Pointers” on page 69

Indirection *

The * (indirection) operator determines the value referred to by the pointer-type operand. The operand cannot be a pointer to an incomplete type. If the operand points to an object, the operation yields an lvalue referring to that object. If the operand points to a function, the result is a function designator in C or, in C++, an lvalue referring to the object to which the operand points. Arrays and functions are converted to pointers.

The type of the operand determines the type of the result. For example, if the operand is a pointer to an **int**, the result has type **int**.

Do not apply the indirection operator to any pointer that contains an address that is not valid, such as `NULL`. The result is not defined.

If `p_to_y` is defined as a pointer to an **int** and `y` as an **int**, the expressions:

```
p_to_y = &y;
*p_to_y = 3;
```

cause the variable `y` to receive the value `3`.

Related References
- "Pointers" on page 69

alignof Operator

The **__alignof__** operator returns the number of bytes used in the alignment of its operand. The language feature is orthogonal to C89 and C99. The operand can be an expression or a parenthesized type identifier. If the operand is an expression representing an lvalue, the number returned by **__alignof__** represents the alignment that the lvalue is known to have. The type of the expression is determined at compile time, but the expression itself is not evaluated. If the operand is a type, the number represents the alignment usually required for the type on the target platform.

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

An **__alignof__** expression has the form:

```
►►──__alignof__──┬─unary_expression─┬──────────────────────────────────────►◄
                 └─(──type-id──)────┘
```

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

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

Related References
- "The aligned Variable Attribute" on page 26

sizeof Operator

The **sizeof** operator yields the size in *bytes* of the operand, which can be an expression or the parenthesized name of a type. A **sizeof** expression has the form:

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

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

The **sizeof** operator applied to a type name yields the amount of memory that would be used by an object of that type, including any internal or trailing padding. The size of any of the three kinds of **char** objects (**unsigned**, **signed**, or plain) is the size of a byte, 1. If the operand is a variable length array type, the operand is evaluated. The **sizeof** operator may not be applied to:
- A bit field
- A function type
- An undefined structure or class
- An incomplete type (such as **void**)

The **sizeof** operator applied to an expression yields the same result as if it had been applied to only the name of the type of the expression. At compile time, the compiler analyzes the expression to determine its type, but does not evaluate it. None of the usual type conversions that occur in the type analysis of the expression are directly attributable to the **sizeof** operator. However, if the operand contains operators that perform conversions, the compiler does take these conversions into consideration in determining the type.

The second line of the following sample causes the usual arithmetic conversions to be performed. Assuming that a **short** uses 2 bytes of storage and an **int** uses 4 bytes,

```
short x; ... sizeof (x)        /* the value of sizeof operator is 2 */
short x; ... sizeof (x + 1)    /* value is 4, result of addition is type int */
```

The result of the expression `x + 1` has type **int** and is equivalent to `sizeof(int)`. The value is also `4` if `x` has type **char**, **short**, or **int** or any enumeration type.

Types cannot be defined in a **sizeof** expression.

In the following example, the compiler is able to evaluate the size at compile time. The operand of **sizeof**, an expression, is not evaluated. The value of `b` is the integer constant 5, from initialization to the end of program run time:

```
#include <stdio.h>

int main(void){
  int b = 5;
  sizeof(b++);
  return 0;
}
```

Except in preprocessor directives, you can use a **sizeof** expression wherever an integral constant is required. One of the most common uses for the **sizeof** operator is to determine the size of objects that are referred to during storage allocation, input, and output functions.

Another use of **sizeof** is in porting code across platforms. You should use the **sizeof** operator to determine the size that a data type represents. For example:

```
sizeof(int);
```

The result of a sizeof expression depends on the type it is applied to.

Operand	Result
An array	The result is the total number of bytes in the array. For example, in an array with 10 elements, the size is equal to 10 times the size of a single element. The compiler does not convert the array to a pointer before evaluating the expression.

typeof Operator

The **typeof** operator returns the type of its argument, which can be an expression or a type. The language feature provides a way to derive the type from an expression. The alternate spelling of the keyword, **__typeof__**, is recommended. Given an expression `e`, **`__typeof__`**`(e)` can be used anywhere a type name is needed, for example in a declaration or in a cast.

The **typeof** operator is an orthogonal language extension provided for handling programs developed with GNU C.

A **typeof** construct is of the form:

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

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

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

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

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

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

you can write

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

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

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

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

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

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

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

Label Value Operator &&

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

Related References
- "Labels as Values" on page 142
- "Computed goto" on page 157

Cast Expressions

The cast operator is used for *explicit type conversions*. This operator has the following form, where *T* is a type, and *expr* is an expression:

(*T* **)** *expr*

It converts the value of *expr* to the type *T*. In C, the result of this operation is not an lvalue.

Cast to a Union Type

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

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

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

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

The output of this example is:

```
U.u2 is 1.234
```

A union cast is also valid as a function argument, part of a constant expression for initialization, and in a compound literal statement.

Binary Expressions

A *binary expression* contains two operands separated by one operator.

Not all binary operators have the same precedence.

All binary operators have left-to-right associativity.

The order in which the operands of most binary operators are evaluated is not specified. To ensure correct results, avoid creating binary expressions that depend on the order in which the compiler evaluates the operands.

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

The following table summarizes the operators for binary expressions:

Precedence and associativity of binary operators

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

Related References
- "Operator Precedence and Associativity" on page 83
- "Arithmetic Conversions" on page 119

Multiplication *

The * (multiplication) operator yields the product of its operands. The operands must have an arithmetic or enumeration type. The result is not an lvalue. The usual arithmetic conversions on the operands are performed.

Because the multiplication operator has both associative and commutative properties, the compiler can rearrange the operands in an expression that contains more than one multiplication operator. For example, the expression:

```
sites * number * cost
```

can be interpreted in any of the following ways:

```
(sites * number) * cost
sites * (number * cost)
(cost * sites) * number
```

Division /

The / (division) operator yields the algebraic quotient of its operands. If both operands are integers, any fractional part (remainder) is discarded. Throwing away the fractional part is often called *truncation toward zero*. The operands must have an arithmetic or enumeration type. The right operand may not be zero: the result is undefined if the right operand evaluates to 0. For example, expression 7 / 4 yields the value 1 (rather than 1.75 or 2). The result is not an lvalue.

The usual arithmetic conversions on the operands are performed.

Remainder %

The % (remainder) operator yields the remainder from the division of the left operand by the right operand. For example, the expression 5 % 3 yields 2. The result is not an lvalue.

Both operands must have an integral or enumeration type. If the right operand evaluates to 0, the result is undefined. If either operand has a negative value, the result is such that the following expression always yields the value of a if b is not 0 and a/b is representable:

```
   ( a / b ) * b + a %b;
```

The usual arithmetic conversions on the operands are performed.

Addition +

The + (addition) operator yields the sum of its operands. Both operands must have an arithmetic type, or one operand must be a pointer to an object type and the other operand must have an integral or enumeration type.

When both operands have an arithmetic type, the usual arithmetic conversions on the operands are performed. The result has the type produced by the conversions on the operands and is not an lvalue.

A pointer to an object in an array can be added to a value having integral type. The result is a pointer of the same type as the pointer operand. The result refers to another element in the array, offset from the original element by the amount of the integral value treated as a subscript. If the resulting pointer points to storage outside the array, other than the first location outside the array, the result is undefined. A pointer to one element past the end of an array cannot be used to access the memory content at that address. The compiler does not provide boundary checking on the pointers. For example, after the addition, `ptr` points to the third element of the array:

```
int array[5];
int *ptr;
ptr = array + 2;
```

Subtraction –

The `-` (subtraction) operator yields the difference of its operands. Both operands must have an arithmetic or enumeration type, or the left operand must have a pointer type and the right operand must have the same pointer type or an integral or enumeration type. You cannot subtract a pointer from an integral value.

When both operands have an arithmetic type, the usual arithmetic conversions on the operands are performed. The result has the type produced by the conversions on the operands and is not an lvalue.

When the left operand is a pointer and the right operand has an integral type, the compiler converts the value of the right to an address offset. The result is a pointer of the same type as the pointer operand.

If both operands are pointers to elements in the same array, the result is the number of objects separating the two addresses. The number is of type **ptrdiff_t**, which is defined in the header file `stddef.h`. Behavior is undefined if the pointers do not refer to objects in the same array.

Bitwise Left and Right Shift << >>

The bitwise shift operators move the bit values of a binary object. The left operand specifies the value to be shifted. The right operand specifies the number of positions that the bits in the value are to be shifted. The result is not an lvalue. Both operands have the same precedence and are left-to-right associative.

Operator	Usage
<<	Indicates the bits are to be shifted to the left.
>>	Indicates the bits are to be shifted to the right.

Each operand must have an integral or enumeration type. The compiler performs integral promotions on the operands, and then the right operand is converted to type **int**. The result has the same type as the left operand (after the arithmetic conversions).

The right operand should not have a negative value or a value that is greater than or equal to the width in bits of the expression being shifted. The result of bitwise shifts on such values is unpredictable.

If the right operand has the value `0`, the result is the value of the left operand (after the usual arithmetic conversions).

The `<<` operator fills vacated bits with zeros. For example, if `left_op` has the value `4019`, the bit pattern (in 16-bit format) of `left_op` is:

```
0000111110110011
```

The expression `left_op << 3` yields:

```
0111110110011000
```

The expression `left_op >> 3` yields:

```
0000000111110110
```

Relational < > <= >=

The relational operators compare two operands and determine the validity of a relationship.

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.

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 result has type **int**.

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 == !=

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

The following table describes the two equality operators:

Operator	Usage
`==`	Indicates whether the value of the left operand is equal to the value of the right operand.
`!=`	Indicates whether the value of the left operand is not equal to the value of the right operand.

Both operands must have arithmetic or enumeration types or be pointers to the same type, or one operand must have a pointer type and the other operand must be a pointer to void or a null pointer. The result is type **int**.

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.
`if (x = 3)`	while is taken to be true because `(x = 3)` evaluates to a nonzero value (3). The expression also assigns the value 3 to x.

Related References
- "Simple Assignment =" on page 111

Bitwise AND &

The `&` (bitwise AND) operator compares each bit of its first operand to the corresponding bit of the second operand. If both bits are 1's, the corresponding bit of the result is set to 1. Otherwise, it sets the corresponding result bit to `0`.

Both operands must have an integral or enumeration type. The usual arithmetic conversions on each operand are performed. The result has the same type as the converted operands.

Because the bitwise AND operator has both associative and commutative properties, the compiler can rearrange the operands in an expression that contains more than one bitwise AND operator.

The following example shows the values of `a`, `b`, and the result of `a & b` represented as 16-bit binary numbers:

bit pattern of `a`	`0000000001011100`
bit pattern of `b`	`0000000000101110`
bit pattern of `a & b`	`0000000000001100`

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

```
   1 & 4 evaluates to 0
while
   1 && 4 evaluates to true
```

Related References
- "Logical AND &&" on page 107

Bitwise Exclusive OR ^

The bitwise exclusive OR operator (in EBCDIC, the ^ symbol is represented by the ¬ symbol) compares each bit of its first operand to the corresponding bit of the second operand. If both bits are 1's or both bits are `0`'s, the corresponding bit of the result is set to `0`. Otherwise, it sets the corresponding result bit to `1`.

Both operands must have an integral or enumeration type. The usual arithmetic conversions on each operand are performed. The result has the same type as the converted operands and is not an lvalue.

Because the bitwise exclusive OR operator has both associative and commutative properties, the compiler can rearrange the operands in an expression that contains more than one bitwise exclusive OR operator. Note that the ^ character can be represented by the trigraph `??'`.

The following example shows the values of a, b, and the result of a ^ b represented as 16-bit binary numbers:

bit pattern of a	0000000001011100
bit pattern of b	0000000000101110
bit pattern of a ^ b	0000000001110010

Related References
- "Trigraph Sequences" on page 11

Bitwise Inclusive OR |

The | (bitwise inclusive OR) operator compares the values (in binary format) of each operand and yields a value whose bit pattern shows which bits in either of the operands has the value 1. If both of the bits are 0, the result of that bit is 0; otherwise, the result is 1.

Both operands must have an integral or enumeration type. The usual arithmetic conversions on each operand are performed. The result has the same type as the converted operands and is not an lvalue.

Because the bitwise inclusive OR operator has both associative and commutative properties, the compiler can rearrange the operands in an expression that contains more than one bitwise inclusive OR operator. Note that the | character can be represented by the trigraph ??!.

The following example shows the values of a, b, and the result of a | b represented as 16-bit binary numbers:

bit pattern of a	0000000001011100
bit pattern of b	0000000000101110
bit pattern of a | b	0000000001111110

Note: The bitwise OR (|) should not be confused with the logical OR (||) operator. For example,

```
   1 | 4 evaluates to 5
while
   1 || 4 evaluates to true
```

Related References
- "Trigraph Sequences" on page 11
- "Logical OR ||" on page 108

Logical AND &&

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`

Related References
- "Bitwise AND &" on page 106

Logical OR ||

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
```

Related References
- "Bitwise Inclusive OR |" on page 107

Conditional Expressions

A *conditional expression* is a compound expression that contains a condition (*operand*$_1$), an expression to be evaluated if the condition evaluates to true (*operand*$_2$), and an expression to be evaluated if the condition has the value false (*operand*$_3$).

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

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

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

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

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

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

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

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

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

Type of Conditional C Expressions

In C, a conditional expression is not an lvalue, nor is its result.

Type of One Operand	Type of Other Operand	Type of Result
Arithmetic	Arithmetic	Arithmetic type after usual arithmetic conversions

Type of One Operand	**Type of Other Operand**	**Type of Result**
Structure or union type	Compatible structure or union type	Structure or union type with all the qualifiers on both operands
void	**void**	**void**
Pointer to compatible type	Pointer to compatible type	Pointer to type with all the qualifiers specified for the type
Pointer to type	`NULL` pointer (the constant `0`)	Pointer to type
Pointer to object or incomplete type	Pointer to **void**	Pointer to **void** with all the qualifiers specified for the type

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);
```

Assignment Expressions

An *assignment expression* stores a value in the object designated by the left operand. There are two types of assignment operators: simple assignment and compound assignment.

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

All assignment operators have the same precedence and have right-to-left associativity.

Simple Assignment =

The simple assignment operator has the following form:

lvalue = *expr*

The operator stores the value of the right operand *expr* in the object designated by the left operand *lvalue*.

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

If the left operand is not a class type, 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.

Related References
- "Lvalues and Rvalues" on page 86
- "Pointers" on page 69
- "Type Qualifiers" on page 61

Compound Assignment

The compound assignment operators consist of a binary operator and the simple assignment operator. They perform the operation of the binary operator on both operands and store the result of that operation into the left operand, which must be a modifiable lvalue.

The following table shows the operand types of compound assignment expressions:

Operator	Left Operand	Right Operand
+= or -=	Arithmetic	Arithmetic
+= or -=	Pointer	Integral type

Operator	Left Operand	Right Operand
*=, /=, 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.

When GNU C language features have been enabled, compound expressions and conditional expressions are allowed as lvalues, provided that their operands are lvalues. The following compound assignment of the compound expression `(a, b)` is legal under GNU C, provided that expression `b`, or more generally, the last expression in the sequence, is an lvalue:

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

Comma Expressions

A *comma expression* contains two operands of any type separated by a comma and has left-to-right associativity. The left operand is fully evaluated, possibly producing side effects, and its value, if there is one, is discarded. The right operand is then evaluated. The type and value of the result of a comma expression are those of its right operand, after the usual unary conversions. In C, the result of a comma expression is not an lvalue. The following statements are equivalent:

```
r = (a,b,...,c);
a; b; r = c;
```

The difference is that the comma operator may be suitable for expression contexts, such as loop control expressions.

Similarly, the address of a compound expression can be taken if the right operand is an lvalue.

```
&(a, b)
a, &b
```

Any number of expressions separated by commas can form a single expression because the comma operator is associative. The use of the comma operator guarantees that the subexpressions will be evaluated in left-to-right order, and the value of the last becomes the value of the entire expression.

In the following example, if `omega` has the value `11`, the expression increments `delta` and assigns the value `3` to `alpha`:

```
alpha = (delta++, omega % 4);
```

A sequence point occurs after the evaluation of the first operand. The value of `delta` is discarded.

For example, the value of the expression:

```
intensity++, shade * increment, rotate(direction);
```

is the value of the expression:

```
rotate(direction)
```

The primary use of the comma operator is to produce side effects in the following situations:
- Calling a function
- Entering or repeating an iteration loop
- Testing a condition
- Other situations where a side effect is required but the result of the expression is not immediately needed

In some contexts where the comma character is used, parentheses are required to avoid ambiguity. For example, the function

```
f(a, (t = 3, t + 2), c);
```

has only three arguments: the value of `a`, the value `5`, and the value of `c`. Other contexts in which parentheses are required are in field-length expressions in structure and union declarator lists, enumeration value expressions in enumeration declarator lists, and initialization expressions in declarations and initializers.

In the previous example, the comma is used to separate the argument expressions in a function invocation. In this context, its use does not guarantee the order of evaluation (left to right) of the function arguments.

The following table gives some examples of the uses of the comma operator:

Statement	**Effects**
`for (i=0; i<2; ++i, f() );`	A **for** statement in which `i` is incremented and `f()` is called at each iteration.

Statement	Effects

`if ( f(), ++i, i>1 )`
`   { /* ... */ }`
	An **if** statement in which function `f()` is called, variable `i` is incremented, and variable `i` is tested against a value. The first two expressions within this comma expression are evaluated before the expression `i>1`. Regardless of the results of the first two expressions, the third is evaluated and its result determines whether the **if** statement is processed.

`func( ( ++a, f(a) ) );`	A function call to `func()` in which `a` is incremented, the resulting value is passed to a function `f()`, and the return value of `f()` is passed to `func()`. The function `func()` is passed only a single argument, because the comma expression is enclosed in parentheses within the function argument list.

Chapter 6. Implicit Type Conversions

An expression `e` of a given type is *implicitly converted* if used in one of the following situations:
- Expression `e` is used as an operand of an arithmetic or logical operation.
- Expression `e` is used as a condition in an **if** statement or an iteration statement (such as a **for** loop). Expression `e` will be converted to **bool** (or **int** in C).
- Expression `e` is used in a **switch** statement. Expression `e` will be converted to an integral type.
- Expression `e` is used in an initialization. This includes the following:
  – An assignment is made to an lvalue that has a different type than `e`.
  – A function is provided an argument value of `e` that has a different type than the parameter.
  – Expression `e` is specified in the **return** statement of a function, and `e` has a different type from the defined return type for the function.

The compiler will allow an implicit conversion of an expression `e` to a type `T` if and only if the compiler would allow the following statement:

```
T var = e;
```

For example when you add values having different data types, both values are first converted to the same type: when a **short int** value and an **int** value are added together, the **short int** value is converted to the **int** type.

You can perform explicit type conversions using one of the cast operators, the function style cast, or the C-style cast.

Integral and Floating-Point Promotions

An *integral promotion* is the conversion of one integral type to another where the second type can hold all possible values of the first type. Certain fundamental types can be used wherever an integer can be used. The following fundamental types can be converted through integral promotion are:
- **char**
- **wchar_t**
- **short int**
- enumerators
- objects of enumeration type
- integer bit fields (both signed and unsigned)

Except for **wchar_t**, if the value cannot be represented by an **int**, the value is converted to an **unsigned int**. For **wchar_t**, if an **int** can represent all the values of the original type, the value is converted to the type that can best represent all the values of the original type. For example, if a **long** can represent all the values, the value is converted to a **long**.

Floating-Point Promotions

You can convert an rvalue of type **float** to an rvalue of type **double**. The value of the expression is unchanged. This conversion is a *floating-point promotion*.

Standard Type Conversions

Many C operators cause *implicit type conversions*, which change the type of an expression. When you add values having different data types, both values are first converted to the same type. For example, when a **short int** value and an **int** value are added together, the **short int** value is converted to the **int** type. It can result in loss of data if the value of the original object is outside the range representable by the shorter type.

Implicit type conversions can occur when:
- An operand is prepared for an arithmetic or logical operation.
- An assignment is made to an lvalue that has a different type than the assigned value.
- A function is provided an argument value that has a different type than the parameter.
- The value specified in the **return** statement of a function has a different type from the defined return type for the function.

Lvalue-to-Rvalue Conversions

If an lvalue appears in a situation in which the compiler expects an rvalue, the compiler converts the lvalue to an rvalue.

An lvalue `e` of a type `T` can be converted to an rvalue if `T` is not a function or array type. The type of `e` after conversion will be `T`. The following table lists exceptions to this:

Situation before conversion	Resulting behavior
`T` is an incomplete type	compile-time error
`e` refers to an uninitialized object	undefined behavior
`e` refers to an object not of type `T`	undefined behavior

Related References
- "Lvalues and Rvalues" on page 86

Boolean Conversions

The conversion of any scalar value to type **_Bool** has a result of 0 if the value compares equal to 0; otherwise the result is 1.

The following is an example of boolean conversions:

```
void f(int* a, int b)
{
  bool d = a;  // false if a == NULL
  bool e = b;  // false if b == 0
}
```

The variable `d` will be **false** if `a` is equal to a null pointer. Otherwise, `d` will be **true**. The variable `e` will be **false** if `b` is equal to zero. Otherwise, `e` will be **true**.

Related References
- "Boolean Variables" on page 41

Integral Conversions

You can convert the following:

- An rvalue of integer type (including signed and unsigned integer types) to another rvalue of integer type
- An rvalue of enumeration type to an rvalue of integer type

If you are converting an integer `a` to an unsigned type, the resulting value `x` is the least unsigned integer such that `a` and `x` are congruent modulo 2^*n*, where n is the number of bits used to represent an unsigned type. If two numbers `a` and `x` are congruent modulo 2^*n*, the following expression is true, where the function `pow(m, n)` returns the value of `m` to the power of `n`:

```
a % pow(2, n) == x % pow(2, n)
```

If you are converting an integer `a` to a signed type, the compiler does not change the resulting value if the new type is large enough to hold `a`. If the new type is not large enough, the behavior is defined by the compiler.

Integer promotions belong to a different category of conversions; they are not integral conversions.

Related References
- "Integer Variables" on page 43

Floating-Point Conversions

You can convert an rvalue of floating-point type to another rvalue of floating-point type.

Floating-point promotions (converting from **float** to **double**) belong to a different category of conversions; they are not floating-point conversions.

Related References
- "Floating-Point Variables" on page 42
- "Integral and Floating-Point Promotions" on page 115

Pointer Conversions

Pointer conversions are performed when pointers are used, including pointer assignment, initialization, and comparison.

Conversions that involve pointers must use an explicit type cast. The exceptions to this rule are the allowable assignment conversions for C pointers. In the following table, a `const`-qualified lvalue cannot be used as a left operand of the assignment.

Table 1. Legal assignment conversions for C pointers

Left operand type	Permitted right operand types
pointer to (object) `T`	the constant `0` a pointer to a type compatible with `T` a pointer to `void` (**void***)
pointer to (function) `F`	the constant `0` a pointer to a function compatible with `F`

The referenced type of the left operand must have the same qualifiers as the right operand. An object pointer may be an incomplete type if the other pointer has type **void***.

C pointers are not necessarily the same size as type **int**. Pointer arguments given to functions should be explicitly cast to ensure that the correct type expected by the function is being passed. The generic object pointer in C is **void***, but there is no generic function pointer.

Conversion to void*

Any pointer to an object of a type `T`, optionally type-qualified, can be converted to **void***, keeping the same **const** or **volatile** qualifications.

The allowable assignment conversions involving **void*** as the left operand are shown in the following table.

*Table 2. Legal assignment conversions in C for void**

Left operand type	Permitted right operand types
(**void***)	the constant `0` a pointer to (object) `T` (**void***)

The object `T` may be an incomplete type.

Null Pointer Constants

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

Array-to-Pointer Conversions

You can convert an lvalue or rvalue with type "array of `N`," where `N` is the type of a single element of the array, to `N*`. The result is a pointer to the initial element of the array. You cannot perform the conversion if the expression is used as the operand of the `&` (address) operator or the **sizeof** operator.

Function-to-Pointer Conversions

You can convert an lvalue that is a function of type `T` to an rvalue that is a pointer to a function of type `T`, except when the expression is used as the operand of the `&` (address) operator, the `()` (function call) operator, or the **sizeof** operator.

Function Argument Conversions

If a function declaration is present and includes declared argument types, the compiler performs type checking. If no function declaration is visible when a function is called, or when an expression appears as an argument in the variable part of a prototype argument list, the compiler performs default argument promotions or converts the value of the expression before passing any arguments to the function. The automatic conversions consist of the following:
- Integral promotions
- Arguments with type **float** are converted to type **double**.

When compiled using a compiler option that allows the GNU C semantics, a function prototype may override a later K&R nonprototype definition. This behavior is illegal in ISO C. Under ISO C, the type of function arguments after automatic conversion must match that of the function prototype.

```
int func(char);     /* Legal in GCC, illegal in ISO C                      */

int func(ch)        /* ch is automatically promoted to int,                */
   char ch;         /* which does not match the prototype argument type char  */
{ return ch == 0;}


int func(float);     /* Legal in GCC, illegal in ISO C                       */

int func(ch)         /* ch is automatically promoted to double,              */
   float ch;         /* which does not match the prototype argument type float */
{ return ch == 0;}
```

Related References
- "Integral and Floating-Point Promotions" on page 115
- "Function Declarations" on page 121

Other Conversions

The void type

By definition, the **void** type has no value. Therefore, it cannot be converted to any other type, and no other value can be converted to **void** by assignment. However, a value can be explicitly cast to **void**.

Structure or union types

No conversions between structure or union types are allowed, except for the following. In C, an assignment conversion between compatible structure or union types is allowed if the right operand is of a type compatible with that of the left operand.

Table 3. Legal assignment conversions in C for structure or union types

Left operand type	Permitted right operand types
a structure or union type	a compatible structure or union type

Enumeration types

In C, when you define a value using the **enum** type specifier, the value is treated as an **int**. Conversions to and from an **enum** value proceed as for the **int** type.

You can convert from an **enum** to any integral type but not from an integral type to an **enum**.

Related References
- "void Type" on page 44
- "Enumerations" on page 57

Arithmetic Conversions

The conversions depend on the specific operator and the type of the operand or operands. However, many operators perform similar conversions on operands of integer and floating-point types. These standard conversions are known as the *arithmetic conversions* because they apply to the types of values ordinarily used in arithmetic.

Arithmetic conversions are used for matching operands of arithmetic operators.

Arithmetic conversion proceeds in the following order:

Operand Type	Conversion
One operand has **long double** type	The other operand is converted to **long double**.
One operand has **double** type	The other operand is converted to **double**.
One operand has **float** type	The other operand is converted to **float**.
One operand has **unsigned long long int** type	The other operand is converted to **unsigned long long int**
One operand has **long long** type.	The other operand is converted to **long long**.
One operand has **unsigned long int** type	The other operand is converted to **unsigned long int**.
One operand has **unsigned int** type and the other operand has **long int** type and the value of the **unsigned int** can be represented in a **long int**	The operand with **unsigned int** type is converted to **long int**.
One operand has **unsigned int** type and the other operand has **long int** type and the value of the **unsigned int** cannot be represented in a **long int**	Both operands are converted to **unsigned long int**.
One operand has **long int** type	The other operand is converted to **long int**.
One operand has **unsigned int** type	The other operand is converted to **unsigned int**.
Both operands have **int** type	The result is type **int**.

Related References
- Chapter 5, “Expressions and Operators,” on page 83
- “Integer Variables” on page 43
- “Floating-Point Variables” on page 42

Chapter 7. Functions

In the context of programming languages, the term *function* means an assemblage of statements used for computing an output value. The word is used less strictly than in mathematics, where it means a set relating input variables *uniquely* to output variables. Functions in C programs may not produce consistent outputs for all inputs, may not produce output at all, or may have side effects. Functions can be understood as user-defined operations, in which the parameters of the parameter list, if any, are the operands.

Functions fall into two categories: those written by you and those provided with the C language implementation. The latter are called *library functions*, since they belong to the library of functions supplied with the compiler.

The result of a function is called its *return value*. The data type of the return value is called the *return type*. A function identifier preceded by its return type and followed by its parameter list is called a *function declaration* or *function prototype*. The term *function body* refers to the statements that represent the actions that the function performs. The body of a function is enclosed in braces, which creates what is called a *function block*. The function return type, followed by its name, parameter list, and body constitute the *function definition*.

The function name followed by the function call operator, (), causes evaluation of the function. If the function has been defined to receive parameters, the values that are to be sent into the function are listed inside the parentheses of the function call operator. These values are the *arguments* for the parameters, and the process just described is called *passing arguments* to the function.

Function Declarations

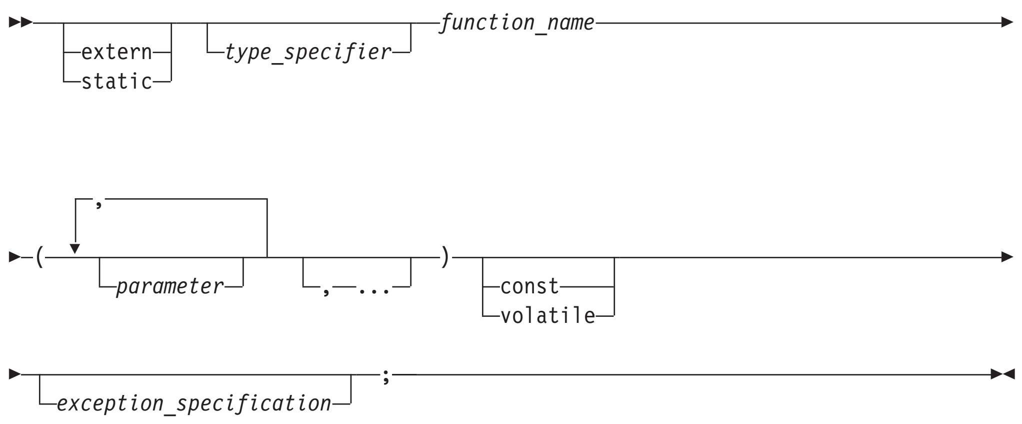
A function *declaration* establishes the name of the function and the number and types of its parameters. A function declaration consists of a return type, a name, and a parameter list. In addition, a function declaration may optionally specify the function's linkage.

A declaration informs the compiler of the format and existence of a function prior to its use. A function can be declared several times in a program, provided that all the declarations agree. Implicit declaration of functions is not allowed: every function must be explicitly declared before it can be called. In C89, if a function is called without an explicit prototype, the compiler provides an implicit declaration. The compiler checks for mismatches between the parameters of a function call and those in the function declaration. The compiler also uses the declaration for argument type checking and argument conversions.

A function *definition* contains a function declaration and the body of the function. A function can only have one definition.

Declarations are typically placed in header files, while function definitions appear in source files.

Function Declarations



A *function argument* is an expression that you use within the parentheses of a function call. A *function parameter* is an object or reference declared within the parenthesis of a function declaration or definition. When you call a function, the arguments are evaluated, and each parameter is initialized with the value of the corresponding argument. The semantics of argument passing are identical to those of assignment.

Some declarations do not name the parameters within the parameter lists; the declarations simply specify the types of parameters and the return values. This is called *prototyping* A function prototype consists of the function return type, the name of the function, and the parameter list. The following example demonstrates this:

```
int func(int,long);
```

Function Return Type

You can define a function to return any type of value, except an array type or a function type. These exclusions must be handled by returning a pointer to the array or function. A function may return a pointer to function, or a pointer to the first element of an array, but it may not return a value that has a type of array or function. To indicate that the function does not return a value, declare it with a return type of **void**.

The return type of a function must be **void** if the return statement does not contain an expression. However, if an expression does appear in the return statement, then the return type of the function cannot be **void**: the compiler converts the return expression as if by assignment to the return type of the function.

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.

Limitations When Declaring a Function

In C, you cannot have an ellipsis as the only argument.

Types cannot be defined in return or argument types. The C compiler allows the following declaration:

```
struct X { int i; };
void print(struct X x);
```

The C compiler will not allow the following declaration of the same function:

```
void print(struct X { int i; } x);   //error
```

This example attempts to declare a function `print()` that takes an object `x` of class `X` as its argument. However, the class definition is not allowed within the argument list.

Related References
- "Type Qualifiers" on page 61

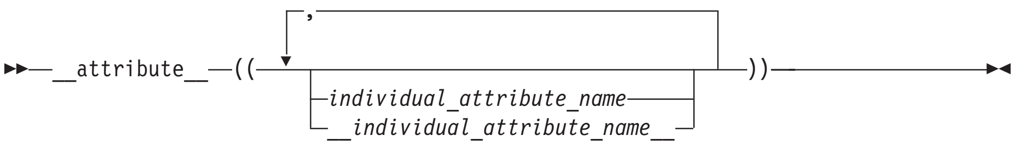
Function Attributes

Function attributes are orthogonal extensions, implemented to enhance the portability of programs developed with GNU C. Specifiable attributes for functions provide explicit ways to help the compiler optimize function calls and to instruct it to check more aspects of the code. Others provide additional functionality.

IBM C 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.

The IBM language extensions for function attributes preserve the GNU C syntax. A function attribute specification using the form __*attribute_name*__ (that is, the function attribute keyword with double underscore characters leading and trailing) reduces the likelihood of a name conflict with a macro of the same name.

The keyword **__attribute__** introduces an attribute specifier. Some of the attributes can also be applied to variables. The syntax is of the general form:



Function attributes are attached to a declarator.

For attributes specified on a function prototype declaration, attaching them to the declarator means placing them after the closing parenthesis of the parameter list.

```
/* Specify the attribute on a function prototype declaration */
void f(int i, int j) __attribute__((individual_attribute_name));
void f(int i, int j) { }
```

Due to ambiguities in parsing old-style parameter declarations, a function definition must have the attribute specification precede the declarator. For example, the following definitions of `foo` show the correct placement:

```
int __attribute__((individual_attribute_name)) foo(int i) { }
int __attribute__((individual_attribute_name)) foo(i,j)
  int i; int j;
{ }
```

Related References
- "Variable Attributes" on page 26
- "Type Attributes" on page 36

The alias Function Attribute

AIX Linux z/OS The **alias** function attribute causes the function declaration to appear in the object file as an alias for another symbol. This language feature provides a technique for coping with duplicate or cumbersome names.

The **alias** function attribute follows the general syntax for function attributes. The following diagram shows the supported forms.

```
►►──__attribute__──((──┬─alias─────┬──(──"original_function_name"──)──))──────────►◄
                       └─__alias__─┘
```

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`:

```
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 References
- "The weak Function Attribute" on page 127

The always_inline Function Attribute

Function attribute `always_inline` instructs the compiler to inline an **inline** function, regardless of whether optimization was specified at compile time. However, the attribute has no effect if the program is compiled at no-opt levels. Specifying this attribute for a function without an **inline** specification also has no effect. The attribute takes precedence over inlining compiler options. The language feature is an orthogonal extension to C89, C99, Standard C++ and C++98, and has been implemented to facilitate porting programs developed with GNU C and C++.

The syntax is shown in the following diagram.

```
►►──__attribute__──((──┬─always_inline─────┬──))──────────────────────────────►◄
                       └─__always_inline__─┘
```

The const Function Attribute

The **const** function attribute allows you to tell the compiler that the function can safely be called fewer times than indicated in the source code. The language feature provides the programmer with an explicit way to help the compiler optimize code by indicating that the function does not examine any values except its arguments and has no effects except for its return value.

The **const** function attribute follows the general syntax for function attributes.

```
►►──__attribute__──((──┬─const─────┬──))──────────────────────────────────────►◄
                       └─__const__─┘
```

The following kinds of functions should not be declared **const**:
- A function with pointer arguments which examines the data pointed to.

- A function that calls a non-**const** function.

See also `#pragma isolated_call` in *XL C Compiler Reference*.

The format Function Attribute

Function attribute `format` provides a way to identify user-defined functions that take format strings as arguments so that calls to these functions will be type-checked against a format string, similar to the way the compiler checks calls to the functions `printf`, `scanf`, `strftime`, and `strfmon` for errors. The feature is an orthogonal extension to C89, C99, Standard C++ and C++98, and has been implemented to facilitate porting applications developed with GNU C and C++.

The syntax is shown in the following diagram. The first argument indicates the archetype for how the format string should be interpreted.

```
                      ┌─,────────────────────────────────────────────────────────────────────┐
                      ↓                                                                      │
►►──__attribute__──((──┬─format─────┬─(──┬─printf──────┬─,─string_index─,─first_to_check─)─┴─))──────────►◄
                       └─__format__─┘    ├─scanf───────┤
                                         ├─strftime────┤
                                         ├─strfmon─────┤
                                         ├─__printf__──┤
                                         ├─__scanf__───┤
                                         ├─__strftime__┤
                                         └─__strfmon__─┘
```

where

string_index
: Is a constant integral expression that specifies which argument in the declaration of the user function is the format string argument. In C++, the minimum value of *string_index* for nonstatic member functions is 2 because the first argument is an implicit **this** argument. This behavior is consistent with that of GNU C++.

first_to_check
: Is a constant integral expression that specifies the first argument to check against the format string. If there are no arguments to check against the format string (that is, diagnostics should only be performed on the format string syntax and semantics), *first_to_check* should have a value of `0`. For `strftime`-style formats, *first_to_check* is required to be `0`.

It is possible to specify multiple `format` attributes on the same function, in which case, all apply.

```
void my_fn(const char* a, const char* b, ...)
      __attribute__((__format__(__printf__,1,0), __format__(__scanf__,2,3)));
```

It is also possible to diagnose the same string for different format styles. All styles are diagnosed.

```
void my_fn(const char* a, const char* b, ...)
      __attribute__((__format__(__printf__,2,3),
                     __format__(__strftime__,2,0),
                     __format__(__scanf__,2,3)));
```

The format_arg Function Attribute

Function attribute `format_arg` provides a way to identify user-defined functions that modify format strings. Once the function is identified, calls to functions like `printf`, `scanf`, `strftime`, or `strfmon`, whose operands are a call to the user-defined function can be checked for errors. The language feature is an orthogonal extension to C89, C99, and Standard C++, and has been implemented to facilitate porting programs developed with GNU C and C++.

The syntax is shown in the following diagram.

```
►►──__attribute__──((──┬─format_arg─────┬──(──string_index──)──))──────────────►◄
                       └─__format_arg__─┘
```

where *string_index* is a constant integral expression that specifies which argument is the format string argument, starting from 1. For non-static member functions in C++, *string_index* starts from 2 because the first parameter is an implicit **this** parameter.

It is possible to specify multiple `format_arg` attributes on the same function, in which case, all apply.

```
extern char* my_dgettext(const char* my_format, const char* my_format2)
          __attribute__((__format_arg__(1))) __attribute__((__format_arg__(2)));

printf(my_dgettext("%","%"));
//printf-style format diagnostics are performed on both "%" strings
```

The noinline Function Attribute

Function attribute `noinline` prevents the function to which it is applied from being inlined, regardless if the function is declared inline or non-inline. The attribute takes precedence over inlining compiler options, the **inline** keyword, and the `always_inline` function attribute. The language feature is an orthogonal extension to C89, C99, Standard C++ and C++98, and has been implemented to facilitate porting programs developed with GNU C and C++.

The syntax is shown in the following diagram.

```
►►──__attribute__──((──┬─noinline─────┬──))────────────────────────────────────►◄
                       └─__noinline__─┘
```

Other than preventing inlining, the attribute does not remove the semantics of inline functions.

The noreturn Function Attribute

The **noreturn** function attribute allows you to indicate to the compiler that the function is not intended to return. The language feature provides the programmer with another explicit way to help the compiler optimize code and to reduce false warnings for uninitialized variables.

The return type of the function should be **void**.

The **noreturn** function attribute follows the general syntax for function attributes.

```
►►──__attribute__──((──┬─noreturn─────┬──))────────────────────────────────────►◄
                       └─__noreturn__─┘
```

Registers saved by the calling function may not necessarily be restored before calling the nonreturning function.

See also `#pragma leaves` in *XL C Compiler Reference*.

The pure Function Attribute

The function attribute **pure** allows you to declare a function that can be called fewer times than what is literally in the source code. Declaring a function with the attribute **pure** indicates that the function has no effect except a return value that depends only on the parameters, global variables, or both. The syntax is the same as that for **const**.

See also `#pragma isolated_call` in *XL C Compiler Reference*.

The weak Function Attribute

AIX Linux The **weak** function attribute causes the symbol resulting from the function declaration to appear in the object file as a weak symbol, rather than a global one. The **weak** attribute can also be applied to variables. The language feature provides the programmer writing library functions with a way to preempt duplicate name errors if the user overrides the function definition in his or her code.

The **weak** function attribute follows the general syntax for function attributes. The following diagram shows the supported forms.

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

Normally, when several relocatable object files are processed, the linker disallows multiple definitions of global symbols with the same name. However, the linker allows a weak definition in the presence of a global symbol with the same name; the weak definition is ignored. Another difference between a global and a weak symbol lies in whether the linker searches archive libraries. To resolve undefined global symbols, the linker searches archive libraries and extracts members that contain definitions; it does not do this to resolve undefined weak symbols.

The following restrictions and limitations apply to weak symbols:

- Weak symbols may not have `static` storage duration.
- Multiple definitions for a weak symbol cannot be provided in the same translation unit. When multiple definitions are present, the linker uses the first weak definition encountered.

Examples of Function Declarations

The following code fragments show several function declarations. The first declares a function `f` that takes two integer arguments and has a return type of **void**:

```
void f(int, int);
```

The following code fragment declares a pointer `p1` to a function that takes a pointer to a constant character and returns an integer:

```
int (*p1) (const char*);
```

The following code fragment declares a function `f1` that takes an integer argument, and returns a pointer to a function that takes an integer argument and returns an integer:

```
int (*f1(int)) (int);
```

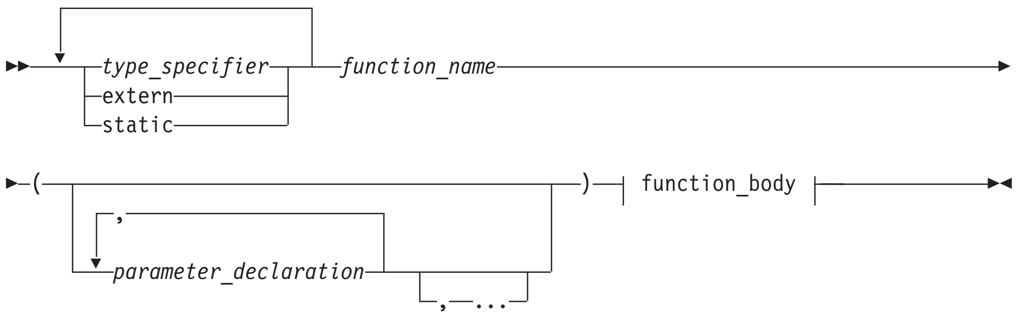
Alternatively, a **typedef** can be used for the complicated return type of function `f1`:

```
typedef int f1_return_type(int);
f1_return_type* f1(int);
```

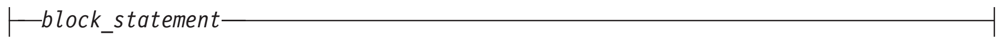
Function Definitions

A *function definition* contains a function declaration and the body of a function.

The syntax for a C function definition is as follows:



function_body:



A function definition contains the following:

- At least one *type specifier*, which determines the type of value that the function returns. For example, the syntax for a function that returns an **unsigned long int** uses three type specifiers.
- An optional *storage class specifier* **extern** or **static**, which determines the scope of the function. If a storage class specifier is not given, the function has external linkage.
- A *function declarator* is the function name followed by a parenthesized list of parameter types and names each parameter that the function expects. In the following function definition, `f(int a, int b)` is the function declarator:
  ```
  int f(int a, int b) {
    return a + b;
  }
  ```
- A *block statement*, which contains data definitions and code.

A function can be called by itself or by other functions. By default, function definitions have external linkage, and can be called by functions defined in other files. A storage class specifier of **static** means that the function name has global scope only, and can be directly invoked only from within the same translation unit.

In C, if a function definition has external linkage and a return type of **int**, calls to the function can be made before it is visible because an implicit declaration of `extern int func();` is assumed. To be compatible with C++, all functions must be declared with prototypes.

If the function does not return a value, use the keyword **void** as the type specifier. If the function does not take any parameters, use the keyword **void** rather than an empty parameter list to indicate that the function is not passed any arguments. In C, a function with an empty parameter list signifies a function that takes an unknown number of parameters; in C++, it means it takes no parameters.

In C, you cannot declare a function as a struct or union member.

Compatibility of Function Declarations

All declarations for a given function must be compatible; that is, the return type is the same and the parameters have the same type.

Compatibility of Function Types

The notion of type compatibility pertains only to C. For two function types to be compatible, the return types must be compatible. If both function types are specified without prototypes, this is the only requirement.

For two functions declared with prototypes, the composite type must meet the following additional requirements:
- If one of the function types has a parameter type list, the composite type is a function prototype with the same parameter type list.
- If both types are function types with parameter lists, then each parameter in the parameter list of the composite is the composite type of the corresponding parameters.

and may use the [*] notation in their sequences of declarator specifiers to specify variable length array types.

If the function declarator is not part of the function definition, the parameters may have incomplete type. The parameters may also specify variable length array types by using the [*] notation in their sequences of declarator specifiers. The following are examples of compatible function prototype declarators:

```
double maximum(int n, int m, double a[n][m]);
double maximum(int n, int m, double a[*][*]);
double maximum(int n, int m, double a[ ][*]);
double maximum(int n, int m, double a[ ][m]);
```

Examples of Function Definitions

The following example is a definition of the function `sum`:

```
int sum(int x,int y)
{
   return(x + y);
}
```

The function `sum` has external linkage, returns an object that has type **int**, and has two parameters of type **int** declared as `x` and `y`. The function body contains a single statement that returns the sum of `x` and `y`.

In the following example, `ary` is an array of two function pointers. Type casting is performed to the values assigned to `ary` for compatibility:

```
#include <stdio.h>

typedef void (*ARYTYPE)();

int func1(void);
void func2(double a);
```

```
int main(void)
{
  double num = 333.3333;
  int retnum;
  ARYTYPE ary[2];
  ary[0]=(ARYTYPE)func1;
  ary[1]=(ARYTYPE)func2;

  retnum=((int (*)())ary[0])();      /*  calls func1  */
  printf("number returned = %i\n", retnum);
  ((void (*)(double))ary[1])(num);   /*  calls func2  */

  return(0);
}

int func1(void)
{
  int number=3;
  return number;
}

void func2(double a)
{
  printf("result of func2 = %f\n", a);
}
```

The following is the output of the above example:

```
number
returned = 3
result of func2 = 333.333300
```

Related References
- "extern Storage Class Specifier" on page 31
- "static Storage Class Specifier" on page 33
- "Type Qualifiers" on page 61

Ellipsis and void

An ellipsis at the end of the parameter specifications is used to specify that a function has a variable number of parameters. The number of parameters is equal to, or greater than, the number of parameter specifications. At least one parameter declaration must come before the ellipsis.

```
int f(int, ...);
```

The comma before the ellipsis as well as a parameter declaration before the ellipsis are both required in C.

Parameter promotions are performed as needed, but no type checking is done on the variable arguments.

You can declare a function with no arguments in two ways:

```
int f(void);
int f();
```

An empty argument declaration list means that the function may take any number or type of parameters.

The type **void** cannot be used as an argument type, although types derived from **void** (such as pointers to **void**) can be used.

In the following example, the function `f()` takes one integer argument and returns no value, while `g()` expects no arguments and returns an integer.

```
void f(int);
int g(void);
```

Examples of Function Definitions

The following example contains a function declarator `i_sort` with `table` declared as a pointer to **int** and `length` declared as type **int**. Note that arrays as parameters are implicitly converted to a pointer to the element type.

```
/**
 ** This example illustrates function definitions.
 ** Note that arrays as parameters are implicitly
 ** converted to a pointer to the type.
 **/

#include <stdio.h>

void i_sort(int table[ ], int length);

int main(void)
{
   int table[ ]={1,5,8,4};
   int length=4;
   printf("length is %d\n",length);
   i_sort(table,length);
}
void i_sort(int table[ ], int length)
{
  int i, j, temp;

  for (i = 0; i < length -1; i++)
    for (j = i + 1; j < length; j++)
      if (table[i] > table[j])
      {
        temp = table[i];
        table[i] = table[j];
        table[j] = temp;
      }
}
```

The following are examples of function declarations (also called *function prototypes*):

```
double square(float x);
int area(int x,int y);
static char *search(char);
```

The following example illustrates how a **typedef** identifier can be used in a function declarator:

```
typedef struct tm_fmt { int minutes;
                      int hours;
                      char am_pm;
                    } struct_t;
long time_seconds(struct_t arrival)
```

The following function `set_date` declares a pointer to a structure of type `date` as a parameter. `date_ptr` has the storage class specifier **register**.

```
void set_date(register struct date *date_ptr)
{
  date_ptr->mon = 12;
  date_ptr->day = 25;
  date_ptr->year = 87;
}
```

C99 requires at least one type specifier for each parameter in a declaration, which reduces the number of situations where the compiler behaves as if an implicit **int** were declared. Prior to C99, the type of `b` or `c` in the declaration of `foo` is ambiguous, and the compiler would assume an implicit **int** for both.

```
int foo( char a, b, c )
{
   /* statements */
}
```

For backward compatibility, some constructs that appear to violate the C99 rule are still allowed. For example, the next definition of `foo` explicitly declares the type for each of the parameters.

```
int foo( int a, char b, int c )
{
   /* statements */
}
```

However, the following definition, which uses an older syntax, is accepted as equivalent, only if `b` and `c` are not referred to in the body of the function.

```
int foo( int a, b, c )
        int a;
{
   /* okay if neither b nor c is used within the function */
}
```

The main() Function

When a program begins running, the system calls the function `main`, which marks the entry point of the program. Every program must have one function named `main`. No other function in the program can be called `main`. A `main` function has one of two forms:

`int main (void)` *block_statement*

`int main (int argc, char ** argv)`*block_statement*

The argument `argc` is the number of command-line arguments passed to the program. The argument `argv` is a pointer to an array of strings, where `argv[0]` is the name you used to run your program from the command-line, `argv[1]` the first argument that you passed to your program, `argv[2]` the second argument, and so on.

By default, `main` has the storage class **extern**.

Arguments to main

The function `main` can be declared with or without parameters.

```
int main(int argc, char *argv[])
```

Although any name can be given to these parameters, they are usually referred to as `argc` and `argv`.

The first parameter, `argc` (argument count), has type **int** and indicates how many arguments were entered on the command line.

The second parameter, `argv` (argument vector), has type array of pointers to **char** array objects. **char** array objects are null-terminated strings.

The value of `argc` indicates the number of pointers in the array `argv`. If a program name is available, the first element in `argv` points to a character array that contains the program name or the invocation name of the program that is being run. If the name cannot be determined, the first element in `argv` points to a null character.

This name is counted as one of the arguments to the function `main`. For example, if only the program name is entered on the command line, `argc` has a value of `1` and `argv[0]` points to the program name.

Regardless of the number of arguments entered on the command line, `argv[argc]` always contains `NULL`.

Example of Arguments to main

The following program `backward` prints the arguments entered on a command line such that the last argument is printed first:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  while (--argc > 0)
    printf("%s ", argv[argc]);
}
```

Invoking this program from a command line with the following:

```
   backward string1 string2
```

gives the following output:

```
   string2 string1
```

The arguments `argc` and `argv` would contain the following values:

Object	**Value**
`argc`	3
`argv[0]`	pointer to string `"backward"`
`argv[1]`	pointer to string `"string1"`
`argv[2]`	pointer to string `"string2"`
`argv[3]`	`NULL`

Note: Be careful when entering mixed case characters on a command line because some environments are not case-sensitive. Also, the exact format of the string pointed to by `argv[0]` is system-dependent.

Calling Functions and Passing Arguments

The arguments of a function call are used to initialize the parameters of the function definition. Array expressions and C function designators as arguments are converted to pointers before the call.

Integral and floating-point promotions will first be done to the values of the arguments before the function is called.

The type of an argument is checked against the type of the corresponding parameter in the function declaration. The size expressions of each variably modified parameter are evaluated on entry to the function. All standard and user-defined type conversions are applied as necessary. The value of each argument expression is converted to the type of the corresponding parameter as if by assignment.

For example:

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

/* Declaration */
extern double root(double, double);

/* Definition */
double root(double value, double base) {
  double temp = exp(log(value)/base);
  return temp;
}

int main(void) {
  int value = 144;
  int base = 2;
  printf("The root is: %f\n", root(value, base));
  return 0;
}
```

The output is `The root is: 12.000000`

In the above example, because the function `root` is expecting arguments of type **double**, the two **int** arguments `value` and `base` are implicitly converted to type **double** when the function is called.

The order in which arguments are evaluated and passed to the function is implementation-defined. For example, the following sequence of statements calls the function `tester`:

```
int x;
x = 1;
tester(x++, x);
```

The call to `tester` in the example may produce different results on different compilers. Depending on the implementation, `x++` may be evaluated first or `x` may be evaluated first. To avoid the ambiguity and have `x++` evaluated first, replace the preceding sequence of statements with the following:

```
int x, y;
x = 1;
y = x++;
tester(y, x);
```

Passing Arguments by Value

If you call a function with an argument that corresponds to a non-pointer parameter, you have passed that argument by value. The parameter is initialized with the value of the argument. You can change the value of the parameter (if that parameter has not been declared **const**) within the scope of the function, but these changes will not affect the value of the argument in the calling function.

The following are examples of passing arguments by value:

The following statement calls the function **printf**, which receives a character string and the return value of the function `sum`, which receives the values of `a` and `b`:

```
printf("sum = %d\n", sum(a,b));
```

The following program passes the value of count to the function increment, which increases the value of the parameter x by 1.

```
/**
 ** An example of passing an argument to a function
 **/

#include <stdio.h>

void increment(int);

int main(void)
{
  int count = 5;

  /* value of count is passed to the function */
  increment(count);
  printf("count = %d\n", count);

  return(0);
}

void increment(int x)
{
  ++x;
  printf("x = %d\n", x);
}
```

The output illustrates that the value of count in main remains unchanged:

```
x = 6
count = 5
```

Related References
- "Function Call Operator ( )" on page 90

Passing Arguments by Reference

Passing by reference refers to a method of passing arguments where the value of an argument in the calling function can be modified in the called function.

To pass an argument by reference, you declare the corresponding parameter with a pointer type.

The following example shows how arguments are passed by reference. Note that pointer parameters are initialized with pointer values when the function is called.

When the function swapnum() is called, the actual values of the variables a and b are exchanged because they are passed by reference. The output is:

```
A is 20 and B is 10
```

You must define the parameters of swapnum() as references if you want the values of the actual arguments to be modified by the function swapnum().

You can modify the values of nonconstant objects through pointer parameters. The following example demonstrates this:

```
#include <stdio.h>

int main(void)
{
  void increment(int *x);
  int count = 5;
  /* address of count is passed to the function */
  increment(&count);
  printf("count = %d\n", count);

  return(0);
}

void increment(int *x)
{
  ++*x;
  printf("*x = %d\n", *x);
}
```

The following is the output of the above code:

```
*x = 6
count = 6
```

The example passes the address of `count` to `increment()`. Function `increment()` increments `count` through the pointer parameter `x`.

Function Return Values

You must return a value from a function unless the function has a return type of **void**.

The return value is specified in a **return** statement. The following code fragment shows a function definition, including the **return** statement:

```
int add(int i, int j)
{
  return i + j; // return statement
}
```

The function `add()` can be called as shown in the following code fragment:

```
int a = 10,
    b = 20;
int answer = add(a, b); // answer is 30
```

In this example, the return statement initializes a variable of the returned type. The variable `answer` is initialized with the **int** value 30. The type of the returned expression is checked against the returned type. All standard and user-defined conversions are performed as necessary.

Each time a function is called, new copies of its variables with automatic storage are created. Because the storage for these automatic variables may be reused after the function has terminated, a pointer or reference to an automatic variable should not be returned.

Related References
- "return Statement" on page 155
- "Value of a return Expression and Function Value" on page 155

Pointers to Functions

A pointer to a function points to the address of the executable code of the function. You can use pointers to call functions and to pass functions as arguments to other functions. You cannot perform pointer arithmetic on pointers to functions.

The type of a pointer to a function is based on both the return type and parameter types of the function.

A declaration of a pointer to a function must have the pointer name in parentheses. The function call operator `()` has a higher precedence than the dereference operator `*`. Without them, the compiler interprets the statement as a function that returns a pointer to a specified return type. For example:

```
int *f(int a);       /* function f returning an int*                  */
int (*g)(int a);     /* pointer g to a function returning an int      */
char (*h)(int, int)  /* h is a function
                          that takes two integer parameters and returns char */
```

In the first declaration, `f` is interpreted as a function that takes an **int** as argument, and returns a pointer to an **int**. In the second declaration, `g` is interpreted as a pointer to a function that takes an **int** argument and that returns an **int**.

Related References
- "Pointer Conversions" on page 117

Inline Functions

An inline function is one for which the compiler copies the code from the function definition directly into the code of the calling function rather than creating a separate set of instructions in memory. Instead of transferring control to and from the function code segment, a modified copy of the function body may be substituted directly for the function call. In this way, the performance overhead of a function call is avoided.

A function is declared inline by using the **inline** function specifier. The **inline** specifier is only a suggestion to the compiler that an inline expansion can be performed; the compiler is free to ignore the suggestion.

The following code fragment shows an inline function definition.

```
inline int add(int i, int j) { return i + j; }
```

The use of the **inline** specifier does not change the meaning of the function. However, the inline expansion of a function may not preserve the order of evaluation of the actual arguments. Inline expansion also does not change the linkage of a function: the linkage is external by default.

In C, any function with internal linkage can be inlined, but a function with external linkage is subject to restriction. The restrictions are as follows:
- If the **inline** keyword is used in the function declaration, then the function definition must appear in the same translation unit.
- An *inline definition* of a function is one in which all of the file-scope declarations for it in the same translation unit include the **inline** specifier without **extern**.
- An inline definition does not provide an external definition for the function: an external definition may appear in another translation unit. The inline definition serves as an alternative to the external definition when called from within the same translation unit. The C99 Standard does not prescribe whether the inline or external definition is used.

In C, an inline definition is distinct from the corresponding external definition and from any other corresponding inline definitions in other translation units.

When source code is compiled allowing language extensions, the behavior of inline functions follows the GNU C semantics. If a function definition has **extern inline** explicitly specified, the compiler uses the **extern inline** definition only for inlining. The behavior resembles macro expansion. If an **extern inline** definition of a function exists in a header file, an external definition for the function without **extern** or **inline** must be available from another file. This definition is used for calls to that function from files that do not include the header file.

The following example illustrates the semantics of **extern inline**. When compiled with the GNU semantics, a noninline function body is not generated for `two()`.

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

   extern inline void two(void){  // GNU C uses this definition only for inlining
      printf("From inline.h\n");
   }

main.c:
   #include "inline.h"

   int main(void){
      void (*pTwo)() = two;
      two();
      (*pTwo)();
   }

two.c:
   #include<stdio.h>

      void two(){
      printf("In two.c\n");
   }
```

The output below shows the results when the first function call to `two` has indeed been inlined.

```
Using the gcc semantics for the inline keyword:
   From inline.h
   In two.c
```

The compiler might still choose not to inline the **extern inline** function `two`, despite the presence of the **inline** function specifier.

Related References
- "extern Storage Class Specifier" on page 31

Nested Functions

A nested function is a function defined inside the definition of another function. It can be defined wherever a variable declaration is permitted, which allows nested functions within nested functions. Within the containing function, the nested function can be declared prior to being defined by using the **auto** keyword. Otherwise, a nested function has internal linkage. The language feature is an orthogonal extension to C89 and C99, implemented to facilitate porting programs developed with GNU C.

A nested function can access all identifiers of the containing function that precede its definition.

Restrictions and limitations

A nested function must not be called after the containing function exits.

A nested function cannot use a **goto** statement to jump to a label in the containing function, or to a local label declared with the **__label__** keyword inherited from the containing function.

Related References
- "Locally Declared Labels" on page 142

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
  - identifier labels
  - **case** labels
  - **default** labels
- expression statements
- block or compound statements
- selection statements
  - **if** statements
  - **switch** statements
- iteration statements
  - **while** statements
  - **do** statements
  - **for** statements
- jump statements
  - **break** statements
  - **continue** statements
  - **return** statements
  - **goto** statements
- declaration statements

Labels

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

Identifier label statements have the following form:

```
►►—identifier—:—statement————————————————————————————————————————►◄
```

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

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 statements have the following form:

```
►►—case—constant_expression—:—statement——————————————————————————►◄
```

Default label statements have the following form:

```
►►—default—:—statement———————————————————————————————————————————►◄
```

Examples of Labels

```
comment_complete : ;           /* null statement label */
test_for_null : if (NULL == pointer)
```

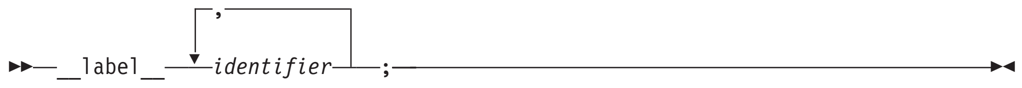
Related References
- "goto Statement" on page 156
- "switch Statement" on page 146

Locally Declared Labels

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

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

The syntax is as follows:



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

Related References
- "Labels" on page 141

Labels as Values

The address of a label defined in the current function or a containing function can be obtained and used as a value wherever a constant of type **void*** is valid. The address is the return value when the label is the operand of the unary operator **&&**. The ability to use the address of label as a value is an orthogonal extension to C99 and C++, implemented to facilitate porting programs developed with GNU C.

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

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

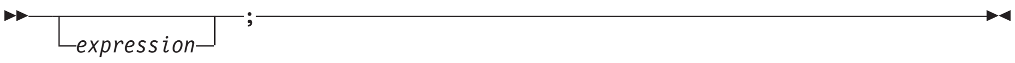
Related References

- “Label Value Operator &&” on page 100
- “Computed goto” on page 157

Expression Statements

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

An expression statement has the form:



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

Examples of Expressions

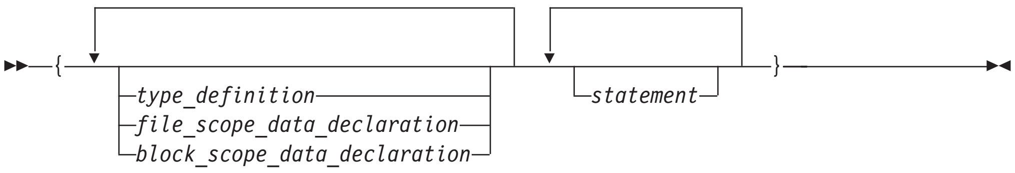
```
printf("Account Number: \n");          /* call to the printf    */
marks = dollars * exch_rate;                /* assignment to marks     */
(difference < 0) ? ++losses : ++gain;  /* conditional increment */
```

Related References
- Chapter 5, “Expressions and Operators,” on page 83

Block Statement

A *block statement*, or *compound statement*, lets you group any number of data definitions, declarations, and statements into one statement. All definitions, declarations, and statements enclosed within a single set of braces are treated as a single statement. You can use a block wherever a single statement is allowed.

A block statement has the form:



At the C89 language level, definitions and declarations must precede any statements.

For C at the C99 language level and for Standard C++ and C++98, declarations and definitions can appear anywhere, mixed in with other code.

A block defines a local scope. If a data object is usable within a block and its identifier is not redefined, all nested blocks can use that data object.

Example of Blocks

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

```
/**
 ** This example shows how data objects change in nested blocks.
 **/
  #include <stdio.h>

  int main(void)
  {
     int x = 1;                    /* Initialize x to 1  */
     int y = 3;

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

     return(0);
  }
```

The program produces the following output:

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

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

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.

An **if** statement has the form:

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

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

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

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

is equivalent to:

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

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

Examples of if Statements

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

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

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

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

The following example shows a nested **if** statement:

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

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

```
if (kegs > 0) {
   if (furlongs > kegs)
      fpk = furlongs/kegs;
}
else
   fpk = 0;
```

The following example shows an **if** statement nested within an **else** clause. This example tests multiple conditions. The tests are made in order of their appearance. If one test evaluates to a nonzero value, a statement runs and the entire **if** statement ends.

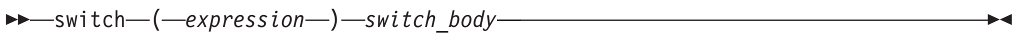
```
if (value > 0)
   ++increase;
else if (value == 0)
   ++break_even;
else
   ++decrease;
```

switch Statement

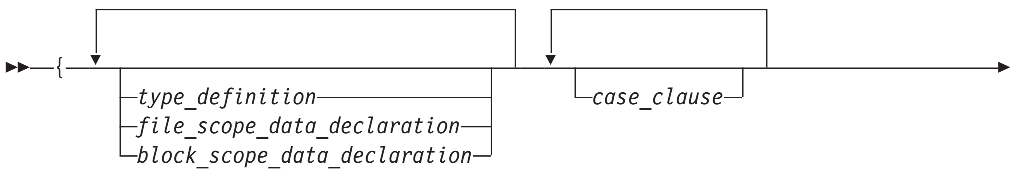
A *switch statement* is a selection statement that lets you transfer control to different statements within the **switch** body depending on the value of the switch expression. The **switch** expression must evaluate to an integral or enumeration value. The body of the **switch** statement contains *case clauses* that consist of
- A **case** label
- An optional **default** label
- A **case** expression
- A list of statements.

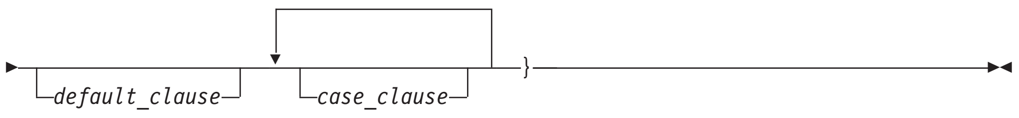
If the value of the **switch** expression equals the value of one of the case expressions, the statements following that case expression are processed. If not, the default label statements, if any, are processed.

A **switch** statement has the form:



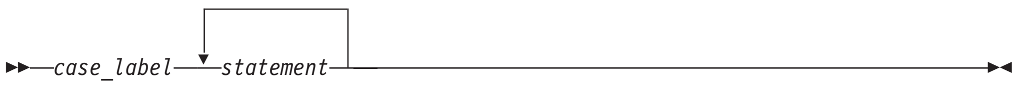
The *switch body* is enclosed in braces and can contain definitions, declarations, *case clauses*, and a *default clause*. Each case clause and default clause can contain statements.



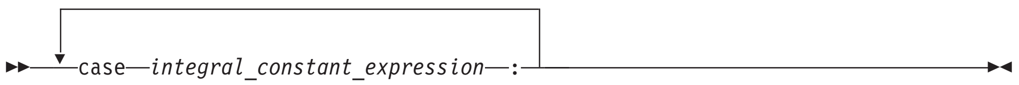


Note: An initializer within a *type_definition*, *file_scope_data_declaration* or *block_scope_data_declaration* is ignored.

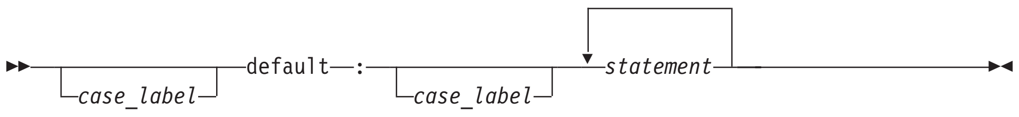
A *case clause* contains a *case label* followed by any number of statements. A case clause has the form:



A *case label* contains the word **case** followed by an integral constant expression and a colon. The value of each integral constant expression must represent a different value; you cannot have duplicate **case** labels. Anywhere you can put one **case** label, you can put multiple **case** labels. A case label has the form:



A *default clause* contains a **default** label followed by one or more statements. You can put a **case** label on either side of the **default** label. A **switch** statement can have only one **default** label. A *default_clause* has the form:



The **switch** statement passes control to the statement following one of the labels or to the statement following the **switch** body. The value of the expression that precedes the **switch** body determines which statement receives control. This expression is called the *switch expression*.

The value of the **switch** expression is compared with the value of the expression in each **case** label. If a matching value is found, control is passed to the statement following the **case** label that contains the matching value. If there is no matching value but there is a **default** label in the **switch** body, control passes to the **default** labelled statement. If no matching value is found, and there is no **default** label anywhere in the **switch** body, no part of the **switch** body is processed.

When control passes to a statement in the **switch** body, control only leaves the **switch** body when a **break** statement is encountered or the last statement in the **switch** body is processed.

If necessary, an integral promotion is performed on the controlling expression, and all expressions in the **case** statements are converted to the same type as the controlling expression. The **switch** expression can also be of class type if there is a single conversion to integral or enumeration type.

Restrictions and Limitations

You can put data definitions at the beginning of the **switch** body, but the compiler does not initialize **auto** and **register** variables at the beginning of a **switch** body. You can have declarations in the body of the **switch** statement.

You cannot use a **switch** statement to jump over initializations.

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.

while Statement

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

A **while** statement has the form:

```
►►──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.

Example of while Statements

In the following program, item[index] triples and is printed out, as long as the value of the expression ++index is less than MAX_INDEX. When ++index evaluates to MAX_INDEX, the **while** statement ends.

```
/**
 ** This example illustrates the while statement.
 **/

#define MAX_INDEX  (sizeof(item) / sizeof(item[0]))
#include <stdio.h>

int main(void)
{
   static int item[ ] = { 12, 55, 62, 85, 102 };
   int index = 0;

   while (index < MAX_INDEX)
   {
      item[index] *= 3;
      printf("item[%d] = %d\n", index, item[index]);
      ++index;
   }

   return(0);
}
```

do Statement

A *do statement* repeatedly runs a statement until the test expression evaluates to 0. Because of the order of processing, the statement is run at least once.

A **do** statement has the form:

```
►►──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.

Example of do Statements

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

```
#include <stdio.h>

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

The following is the output of the above example:

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

for Statement

A *for statement* lets you do the following:

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

A **for** statement has the form:

```
►►──for──(──┬─────────────┬──;──┬─────────────┬──;──┬─────────────┬──)──────────►
            └─expression1─┘     └─expression2─┘     └─expression3─┘

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

expression1 Is the *initialization expression*. It is evaluated only before the *statement* is processed for the first time. You can use this expression to initialize a variable. If you do not want to evaluate an expression prior to the first iteration of the statement, you can omit this expression.

expression2 Is the *conditional expression*. It is evaluated before each iteration of the *statement*.

It must evaluate to an 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.

You can also use *expression1* to declare a variable as well as initialize it. If you declare a variable in this expression, or anywhere else in *statement*, that variable goes out of scope at the end of the **for** loop.

Examples of for Statements

The following **for** statement prints the value of `count` `20` times. The **for** statement initially sets the value of `count` to `1`. After each iteration of the statement, `count` is incremented.

```
int count;
for (count = 1; count <= 20; count++)
   printf("count = %d\n", count);
```

The following sequence of statements accomplishes the same task. Note the use of the **while** statement instead of the **for** statement.

```
int count = 1;
while (count <= 20)
{
   printf("count = %d\n", count);
   count++;
}
```

The following **for** statement does not contain an initialization expression:

```
for (; index > 10; --index)
{
   list[index] = var1 + var2;
   printf("list[%d] = %d\n", index,
   list[index]);
}
```

The following **for** statement will continue running until `scanf` receives the letter `e`:

```
for (;;)
{
   scanf("%c", &letter);
   if (letter == '\n')
      continue;
   if (letter == 'e')
      break;
   printf("You entered the letter %c\n", letter);
}
```

The following **for** statement contains multiple initializations and increments. The comma operator makes this construction possible. The first comma in the **for** expression is a punctuator for a declaration. It declares and initializes two integers, `i` and `j`. The second comma, a comma operator, allows both `i` and `j` to be incremented at each step through the loop.

```
for (int i = 0,
j = 50; i < 10; ++i, j += 50)
{
   cout << "i = " << i << "and j = " << j
   << endl;
}
```

The following example shows a nested **for** statement. It prints the values of an array having the dimensions `[5][3]`.

```
for (row = 0; row < 5; row++)
   for (column = 0; column < 3; column++)
      printf("%d\n",
      table[row][column]);
```

The outer statement is processed as long as the value of `row` is less than 5. Each time the outer **for** statement is executed, the inner **for** statement sets the initial value of `column` to zero and the statement of the inner **for** statement is executed 3 times. The inner statement is executed as long as the value of `column` is less than 3.

break Statement

A *break statement* lets you end an *iterative* (**do**, **for**, or **while**) statement or a **switch** statement and exit from it at any point other than the logical end. A **break** may only appear on one of these statements.

A **break** statement has the form:

```
►►──break──;──────────────────────────────────────────────────────────►◄
```

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

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

continue Statement

A *continue statement* ends the current iteration of a loop. Program control is passed from the **continue** statement to the end of the loop body.

A **continue** statement has the form:

```
►►──continue──;───────────────────────────────────────────────────────►◄
```

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

The **continue** statement ends the processing of the action part of an iterative statement and moves control to the loop continuation portion of the statement. For example, if the iterative statement is a **for** statement, control moves to the third expression in the condition part of the statement, then to the second expression (the test) in the condition part of the statement.

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

Examples of continue Statements

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

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

```
#include <stdio.h>
#define  SIZE  5

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

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

   return(0);
}
```

The program produces the following output:

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

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

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

#include <stdio.h>
#define  SIZE  3

int main(void)
{
   static char *strings[SIZE] = { "ab", "c5d", "e5" };
   int i;
   int letter_count = 0;
   char *pointer;
   for (i = 0; i < SIZE; i++)             /* for each string   */
                                           /* for each each character */
      for (pointer = strings[i]; *pointer != '\0';
      ++pointer)
      {                                  /* if a number      */
         if (*pointer >= '0' && *pointer <= '9')
            continue;
         letter_count++;
      }
   printf("letter count = %d\n", letter_count);

   return(0);
}
```

The program produces the following output:

```
letter count = 5
```

return Statement

A *return statement* ends the processing of the current function and returns control to the caller of the function.

A **return** statement has one of two forms:

```
►►──return──┬────────────┬──;────────────────────────────────────────────►◄
            └─expression─┘
```

A value-returning function must include an *expression* in the **return** statement. A function with a return type is `void` cannot contain an *expression* in its return statement.

For a function of return type `void`, a return statement is not strictly necessary. If the end of such a function is reached without encountering a **return** statement, control is passed to the caller as if a **return** statement without an expression were encountered. In other words, an implicit return takes place upon completion of the final statement, and control automatically returns to the calling function. A function can contain multiple **return** statements. For example:

```
void copy( int *a, int *b, int c)
{
   /* Copy array a into b, assuming both arrays are the same size */

   if (!a || !b)       /* if either pointer is 0, return */
      return;

   if (a == b)         /* if both parameters refer */
      return;          /*    to same array, return */

   if (c == 0)         /* nothing to copy */
      return;

   for (int i = 0; i < c; ++i;) /* do the copying */
      b[i] = a[1];
                       /* implicit return */
}
```

In this example, the **return** statement is used to cause a premature termination of the function, similar to a **break** statement.

An expression appearing in a **return** statement is converted to the return type of the function in which the statement appears. If no implicit conversion is possible, the **return** statement is invalid.

Value of a return Expression and Function Value

If an expression is present on a **return** statement, the value of the expression is returned to the caller. If the data type of the expression is different from the function return type, conversion of the return value takes place as if the value of the expression were assigned to an object with the same function return type.

The value of the **return** statement for a function of return type `void` means that the function does not return a value. If an expression is not given on a **return** statement in a function declared with a non-`void` return type, the complier issues an error message.

You cannot use a **return** statement with an expression when the function is declared as returning type **void**.

Examples of return Statements

```
return;             /* Returns no value           */
return result;      /* Returns the value of result */
return 1;           /* Returns the value 1         */
return (x * x);     /* Returns the value of x * x  */
```

The following function searches through an array of integers to determine if a match exists for the variable `number`. If a match exists, the function `match` returns the value of `i`. If a match does not exist, the function `match` returns the value `-1` (negative one).

```
int match(int number, int array[ ], int n)
{
   int i;

   for (i = 0; i < n; i++)
      if (number == array[i])
         return (i);
   return(-1);
}
```

goto Statement

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

A **goto** statement has the form:

```
►►──goto──label_identifier──;──────────────────────────────────────►◄
```

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

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

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

A **goto** statement is allowed to jump within the scope of a variable length array, but not past any declarations of objects with variably modified types.

Example of goto Statements

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

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

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

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

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

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

Computed goto

A computed goto is a goto statement for which the target is a label from the same function. The address of the label is a constant of type **void***, and is obtained by applying the unary label value operator **&&** to the label. The target of a computed goto is known at run time, and all computed goto statements from the same function will have the same targets. The language feature is an orthogonal extension to C99 and C++, implemented to facilitate porting programs developed with GNU C.

A computed goto is of the form

```
►►──goto──*expression──;──────────────────────────────────────────────►◄
```

where **expression* is an expression of type **void***.

Related References
- "Labels as Values" on page 142
- "Label Value Operator &&" on page 100

Null Statement

The *null statement* performs no operation. It has the form:

```
►►──;─────────────────────────────────────────────────────────────────►◄
```

A null statement can hold the label of a labeled statement or complete the syntax of an iterative statement.

Examples of Null Statements

The following example initializes the elements of the array price. Because the initializations occur within the **for** expressions, a statement is only needed to finish the **for** syntax; no operations are required.

```
for (i = 0; i < 3; price[i++] = 0)
   ;
```

Null Statements

A null statement can be used when a label is needed before the end of a block statement. For example:

```
void func(void) {
  if (error_detected)
    goto depart;
  /* further processing */
  depart: ;  /* null statement required */
}
```

Chapter 9. Preprocessor Directives

The preprocessor is a program that is invoked by the compiler to process code before compilation. Commands for that program, known as *directives*, are lines of the source file beginning with the character #, which distinguishes them from lines of source program text. The effect of each preprocessor directive is a change to the text of the source code, and the result is a new source code file, which does not contain the directives. The preprocessed source code, an intermediate file, must be a valid C program, because it becomes the input to the compiler.

The syntax of preprocessor directives is independent of, but similar to, the syntax of the rest of the language, and the lexical conventions of the preprocessor differ from those of the compiler. The preprocessor recognizes the normal C tokens, as well as other characters that enable the preprocessor to recognize file names, the presence and absence of white space, and the location of end-of-line markers.

Preprocessor directives and the related subject of macro expansion are discussed in this section. After an overview of preprocessor directives, the topics covered include textual macros, file inclusion, ISO standard and predefined macro names, conditional compilation directives, and pragmas.

Preprocessor Overview

Preprocessing is a preliminary operation on C files before they are passed to the compiler. It allows you to do the following:
- Replace tokens in the current file with specified replacement tokens
- Imbed files within the current file
- Conditionally compile sections of the current file
- Generate diagnostic messages
- Change the line number of the next line of source and change the file name of the current file
- Apply machine-specific rules to specified sections of code

A *token* is a series of characters delimited by white space. The only white space allowed on a preprocessor directive is the space, horizontal tab, vertical tab, form feed, and comments. The new-line character can also separate preprocessor tokens.

The preprocessed source program file must be a valid C program.

The preprocessor is controlled by the following directives:

`#define`	Defines a macro.
`#undef`	Removes a preprocessor macro definition.
`#error`	Defines text for a compile-time error message.
`#include`	Inserts text from another source file.
`#if`	Conditionally suppresses portions of source code, depending on the result of a constant expression.
`#ifdef`	Conditionally includes source text if a macro name is defined.
`#ifndef`	Conditionally includes source text if a macro name is not defined.
`#else`	Conditionally includes source text if the previous **#if**, **#ifdef**, **#ifndef**, or **#elif** test fails.
`#elif`	Conditionally includes source text if the previous **#if**, **#ifdef**, **#ifndef**, or **#elif** test fails, depending on the value of a constant expression.

`#endif`	Ends conditional text.
`#line`	Supplies a line number for compiler messages.
`#pragma`	Specifies implementation-defined instructions to the compiler.

Preprocessor Directive Format

Preprocessor directives begin with the # token followed by a preprocessor keyword. The # token must appear as the first character that is not white space on a line. The # is not part of the directive name and can be separated from the name with white spaces.

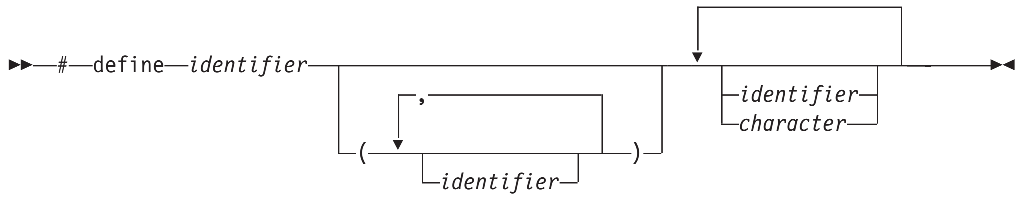
A preprocessor directive ends at the new-line character unless the last character of the line is the \ (backslash) character. If the \ character appears as the last character in the preprocessor line, the preprocessor interprets the \ and the new-line character as a continuation marker. The preprocessor deletes the \ (and the following new-line character) and splices the physical source lines into continuous logical lines. White space is allowed between backslash and the end of line character or the physical end of record. However,this white space is usually not visible during editing.

Except for some **#pragma** directives, preprocessor directives can appear anywhere in a program.

Macro Definition and Expansion (#define)

A *preprocessor define directive* directs the preprocessor to replace all subsequent occurrences of a macro with specified replacement tokens.

A preprocessor **#define** directive has the form:



The **#define** directive can contain an object-like definition or a function-like definition.

#define versus `const`

- The **#define** directive can be used to create a name for a numerical, character, or string constant, whereas a `const` object of any type can be declared.
- A `const` object is subject to the scoping rules for variables, whereas a constant created using **#define** is not.
- Unlike a `const` object, the value of a macro does not appear in the intermediate source code used by the compiler because they are expanded inline. The inline expansion makes the macro value unavailable to the debugger.
- A macro can be used in a constant expression, such as an array bound, whereas a `const` object cannot.

Related References
- "Object-Like Macros" on page 161

- "Function-Like Macros"
- "The const Type Qualifier" on page 62

Object-Like Macros

An *object-like macro definition* replaces a single identifier with the specified replacement tokens. The following object-like definition causes the preprocessor to replace all subsequent instances of the identifier `COUNT` with the constant `1000` :

```
#define COUNT 1000
```

If the statement

```
int arry[COUNT];
```

appears after this definition and in the same file as the definition, the preprocessor would change the statement to

```
int arry[1000];
```

in the output of the preprocessor.

Other definitions can make reference to the identifier `COUNT`:

```
#define MAX_COUNT COUNT + 100
```

The preprocessor replaces each subsequent occurrence of `MAX_COUNT` with `COUNT + 100`, which the preprocessor then replaces with `1000 + 100`.

If a number that is partially built by a macro expansion is produced, the preprocessor does not consider the result to be a single value. For example, the following will not result in the value 10.2 but in a syntax error.

```
#define a 10
a.2
```

Identifiers that are partially built from a macro expansion may not be produced. Therefore, the following example contains two identifiers and results in a syntax error:

```
#define d efg
abcd
```

Function-Like Macros

More complex than object-like macros, a function-like macro definition declares the names of formal parameters within parentheses, separated by commas. An empty formal parameter list is legal: such a macro can be used to simulate a function that takes no arguments. C adds support for function-like macros with a variable number of arguments.

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.

Example of #define Directives

The following program contains two macro definitions and a macro invocation that refers to both of the defined macros:

```
/**
 ** This example illustrates #define directives.
 **/

#include <stdio.h>

#define SQR(s)  ((s) * (s))
#define PRNT(a,b) \
  printf("value 1 = %d\n", a); \
  printf("value 2 = %d\n", b) ;
```

```
int main(void)
{
  int x = 2;
  int y = 3;

     PRNT(SQR(x),y);

  return(0);
}
```

After being interpreted by the preprocessor, this program is replaced by code equivalent to the following:

```
#include <stdio.h>

int main(void)
{
  int x = 2;
  int y = 3;

     printf("value 1 = %d\n", ( (x) * (x) ) );
     printf("value 2 = %d\n", y);

  return(0);
}
```

This program produces the following output:

```
value 1 = 4
value 2 = 3
```

Variadic Macro Extensions

Variadic macro extensions refer to two extensions to C99 related to macros with variable number of arguments. One extension is a mechanism for renaming the variable argument identifier from `__VA_ARGS__` to a user-defined identifier. This extension is orthogonal to C99. The other extension provides a way to remove the dangling comma in a variadic macro when no variable arguments are specified. This extension is non-orthogonal. Both extensions have been implemented to facilitate porting programs developed with GNU C and C++.

An Identifier Instead of `__VA_ARGS__`

The following examples demonstrate the use of an identifier in place of `__VA_ARGS__`. The first definition of the macro `debug` exemplifies the usual usage of `__VA_ARGS__`. The second definition shows the use of the identifier `args` in place of `__VA_ARGS__`.

```
#define debug1(format, ...)  printf(format, __VA_ARGS__)
#define debug2(format, args ...)  printf(format, args)
```

Invocation	Result of Macro Expansion
`debug1("Hello %s\n","World");`	`printf("Hello %s\n","World");`
`debug2("Hello %s\n","World");`	`printf("Hello %s\n","World");`

Trailing Comma Removal

The preprocessor removes the trailing comma if the variable arguments to a function macro are omitted or empty and the comma followed by ## precedes the variable argument identifier in the function macro definition.

Scope of Macro Names (#undef)

A *preprocessor undef directive* causes the preprocessor to end the scope of a preprocessor definition.

A preprocessor **#undef** directive has the form:

```
►►──#──undef──identifier──────────────────────────────────────────────────►◄
```

If the identifier is not currently defined as a macro, **#undef** is ignored.

Example of #undef Directives

The following directives define BUFFER and SQR:

```
#define BUFFER 512
#define SQR(x) ((x) * (x))
```

The following directives nullify these definitions:

```
#undef BUFFER
#undef SQR
```

Any occurrences of the identifiers BUFFER and SQR that follow these **#undef** directives are not replaced with any replacement tokens. Once the definition of a macro has been removed by an **#undef** directive, the identifier can be used in a new **#define** directive.

# Operator

The # (single number sign) operator converts a parameter of a function-like macro into a character string literal. For example, if macro ABC is defined using the following directive:

```
   #define ABC(x)   #x
```

all subsequent invocations of the macro ABC would be expanded into a character string literal containing the argument passed to ABC. For example:

Invocation	Result of Macro Expansion
ABC(1)	"1"
ABC(Hello there)	"Hello there"

The # operator should not be confused with the null directive.

Use the # operator in a function-like macro definition according to the following rules:
- A parameter following # operator in a function- like macro is converted into a character string literal containing the argument passed to the macro.
- White-space characters that appear before or after the argument passed to the macro are deleted.
- Multiple white-space characters imbedded within the argument passed to the macro are replaced by a single space character.
- If the argument passed to the macro contains a string literal and if a \ (backslash) character appears within the literal, a second \ character is inserted before the original \ when the macro is expanded.

- If the argument passed to the macro contains a " (double quotation mark) character, a \ character is inserted before the " when the macro is expanded.
- The conversion of an argument into a string literal occurs before macro expansion on that argument.
- If more than one ## operator or # operator appears in the replacement list of a macro definition, the order of evaluation of the operators is not defined.
- If the result of the macro expansion is not a valid character string literal, the behavior is undefined.

Example of the # Operator

The following examples demonstrate the use of the # operator:

```
   #define STR(x)        #x
   #define XSTR(x)       STR(x)
   #define ONE           1
```

Invocation	Result of Macro Expansion
`STR(\n "\n" '\n')`	`"\n \"\\n\" '\\n'"`
`STR(ONE)`	"ONE"
`XSTR(ONE)`	"1"
`XSTR("hello")`	"\"hello\""

Macro Concatenation with the ## Operator

The ## (double number sign) operator concatenates two tokens in a macro invocation (text and/or arguments) given in a macro definition.

If a macro XY was defined using the following directive:

```
   #define XY(x,y)    x##y
```

the last token of the argument for x is concatenated with the first token of the argument for y.

Use the ## operator according to the following rules:

- The ## operator cannot be the very first or very last item in the replacement list of a macro definition.
- The last token of the item in front of the ## operator is concatenated with first token of the item following the ## operator.
- Concatenation takes place before any macros in arguments are expanded.
- If the result of a concatenation is a valid macro name, it is available for further replacement even if it appears in a context in which it would not normally be available.
- If more than one ## operator and/or # operator appears in the replacement list of a macro definition, the order of evaluation of the operators is not defined.

Examples of the ## Operator

The following examples demonstrate the use of the ## operator:

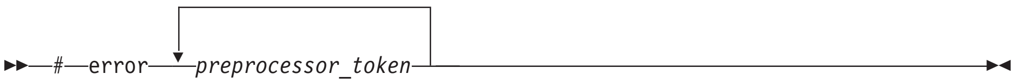
```
#define ArgArg(x, y)          x##y
#define ArgText(x)            x##TEXT
#define TextArg(x)            TEXT##x
#define TextText              TEXT##text
#define Jitter                1
#define bug                   2
#define Jitterbug             3
```

Invocation	Result of Macro Expansion
`ArgArg(lady, bug)`	`"ladybug"`
`ArgText(con)`	`"conTEXT"`
`TextArg(book)`	`"TEXTbook"`
`TextText`	`"TEXTtext"`
`ArgArg(Jitter, bug)`	`3`

Preprocessor Error Directive (#error)

A *preprocessor error directive* causes the preprocessor to generate an error message and causes the compilation to fail.

A **#error** directive has the form:



The **#error** directive is often used in the **#else** portion of a **#if–#elif–#else** construct, as a safety check during compilation. For example, **#error** directives in the source file can prevent code generation if a section of the program is reached that should be bypassed.

For example, the directive

```
#define BUFFER_SIZE 255

#if BUFFER_SIZE < 256
#error "BUFFER_SIZE is too small."
#endif
```

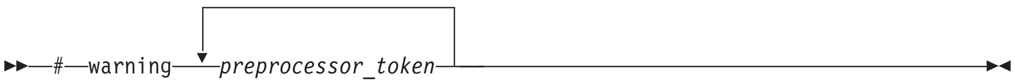
generates the error message:

```
BUFFER_SIZE is too small.
```

Preprocessor Warning Directive (#warning)

A *preprocessor warning directive* causes the preprocessor to generate a warning message but allows compilation to continue. The argument to **#warning** is not subject to macro expansion.

A **#warning** directive has the form:



The preprocessor **#warning** directive is an orthogonal language extension provided to facilitate handling programs developed with GNU C. The IBM implementation preserves multiple white spaces.

File Inclusion (#include)

A *preprocessor include directive* causes the preprocessor to replace the directive with the contents of the specified file.

A preprocessor **#include** directive has the form:

```
►►──#──include──┬─"──file_name──"───┬────────────────────────────────────────►◄
                ├─<──file_name──>───┤
                ├─<──header_name──>─┤
                └─identifiers───────┘
```

In all C implementations, the preprocessor resolves macros contained in an **#include** directive. After macro replacement, the resulting token sequence must consist of a file name enclosed in either double quotation marks or the characters < and >.

For example:

```
#define MONTH <july.h>
#include MONTH
```

If the file name is enclosed in double quotation marks, for example:

```
#include "payroll.h"
```

the preprocessor treats it as a user-defined file, and searches for the file in a manner defined by the preprocessor.

If the file name is enclosed in angle brackets, for example:

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

it is treated as a system-defined file, and the preprocessor searches for the file in a manner defined by the preprocessor.

The new-line and > characters cannot appear in a file name delimited by < and >. The new-line and " (double quotation marks) character cannot appear in a file name delimited by " and ", although > can.

Declarations that are used by several files can be placed in one file and included with **#include** in each file that uses them. For example, the following file `defs.h` contains several definitions and an inclusion of an additional file of declarations:

```
/* defs.h */
#define TRUE 1
#define FALSE 0
#define BUFFERSIZE 512
#define MAX_ROW 66
#define MAX_COLUMN 80
int hour;
int min;
int sec;
#include "mydefs.h"
```

You can embed the definitions that appear in `defs.h` with the following directive:

```
#include "defs.h"
```

In the following example, a **#define** combines several preprocessor macros to define a macro that represents the name of the C standard I/O header file. A **#include** makes the header file available to the program.

```
#define C_IO_HEADER <stdio.h>

/* The following is equivalent to:
 *   #include <stdio.h>
 */

#include C_IO_HEADER
```

Specialized File Inclusion (#include_next)

The preprocessor directive **#include_next** instructs the preprocessor to continue searching for the specified file name, and to include the subsequent instance encountered after the current directory. The syntax of the directive is similar to that of **#include**.

The language feature is an orthogonal extension to C. It extends the techniques available to address the issue of duplicate file names among applications and shared libraries.

ISO 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.

Macro Name **Description**

`__DATE__` A character string literal containing the date when the source file was compiled.

The value of `__DATE__` changes as the compiler processes any include files that are part of your source program. The date is in the form:

```
"Mmm dd yyyy"
```

where:

`Mmm`	Represents the month in an abbreviated form (`Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, or `Dec`).
`dd`	Represents the day. If the day is less than 10, the first `d` is a blank character.
`yyyy`	Represents the year.

`__FILE__` A character string literal containing the name of the source file.

The value of `__FILE__` changes as the compiler processes include files that are part of your source program. It can be set with the **#line** directive.

`__LINE__` An integer representing the current source line number.

The value of `__LINE__` changes during compilation as the compiler processes subsequent lines of your source program. It can be set with the **#line** directive.

`__STDC__` For C, the integer 1 (one) indicates that the C compiler supports the ISO standard. If you set the language level to anything other than ANSI, 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__	The value of this C99 macro is 1, indicating that the C compiler is a hosted implementation.
__STDC_VERSION__	The integer constant of type **long int**: 199409L for the C89 language level, 199901L for C99.
__TIME__	A character string literal containing the time when the source file was compiled. The value of __TIME__ changes as the compiler processes any include files that are part of your source program. The time is in the form: "hh:mm:ss" where: hh Represents the hour. mm Represents the minutes. ss Represents the seconds.

In addition to the predefined macros required by the language standard, the predefined macro __IBMC__ indicates the level of the C compiler.

The value is an integer of the form VRM, where

V	Represents the version number.
R	Represents the release number.
M	Represents the modification number.

Related References
- “Line Control (#line)” on page 174
- “Object-Like Macros” on page 161

Conditional Compilation Directives

A *preprocessor conditional compilation directive* causes the preprocessor to conditionally suppress the compilation of portions of source code. These directives test a constant expression or an identifier to determine which tokens the preprocessor should pass on to the compiler and which tokens should be bypassed during preprocessing. The directives are:
- **#if**
- **#ifdef**
- **#else**
- **#ifndef**
- **#elif**
- **#endif**

The preprocessor conditional compilation directive spans several lines:
- The condition specification line (beginning with **#if**, **#ifdef**, or **#ifndef**)
- Lines containing code that the preprocessor passes on to the compiler if the condition evaluates to a nonzero value (optional)
- The **#elif** line (optional)
- Lines containing code that the preprocessor passes on to the compiler if the condition evaluates to a nonzero value (optional)

- The **#else** line (optional)
- Lines containing code that the preprocessor passes on to the compiler if the condition evaluates to zero (optional)
- The preprocessor **#endif** directive

For each **#if**, **#ifdef**, and **#ifndef** directive, there are zero or more **#elif** directives, zero or one **#else** directive, and one matching **#endif** directive. All the matching directives are considered to be at the same nesting level.

You can nest conditional compilation directives. In the following directives, the first **#else** is matched with the **#if** directive.

```
#ifdef MACNAME
                 /*  tokens added if MACNAME is defined */
#   if TEST <=10
                 /* tokens added if MACNAME is defined and TEST <= 10 */
#   else
                 /* tokens added if MACNAME is defined and TEST >  10 */
#   endif
#else
                 /*  tokens added if MACNAME is not defined */
#endif
```

Each directive controls the block immediately following it. A block consists of all the tokens starting on the line following the directive and ending at the next conditional compilation directive at the same nesting level.

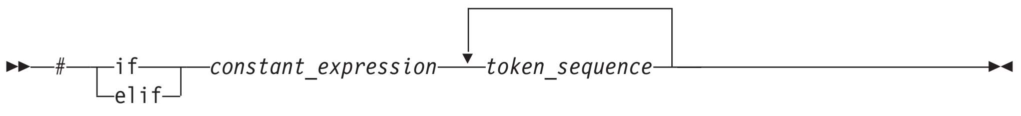
Each directive is processed in the order in which it is encountered. If an expression evaluates to zero, the block following the directive is ignored.

When a block following a preprocessor directive is to be ignored, the tokens are examined only to identify preprocessor directives within that block so that the conditional nesting level can be determined. All tokens other than the name of the directive are ignored.

Only the first block whose expression is nonzero is processed. The remaining blocks at that nesting level are ignored. If none of the blocks at that nesting level has been processed and there is a **#else** directive, the block following the **#else** directive is processed. If none of the blocks at that nesting level has been processed and there is no **#else** directive, the entire nesting level is ignored.

#if, #elif

The **#if** and **#elif** directives compare the value of *constant_expression* to zero:



If the constant expression evaluates to a nonzero value, the lines of code that immediately follow the condition are passed on to the compiler.

If the expression evaluates to zero and the conditional compilation directive contains a preprocessor **#elif** directive, the source text located between the **#elif** and the next **#elif** or preprocessor **#else** directive is selected by the preprocessor to be passed on to the compiler. The **#elif** directive cannot appear after the preprocessor **#else** directive.

All macros are expanded, any `defined()` expressions are processed and all remaining identifiers are replaced with the token `0`.

The *constant_expression* that is tested must be integer constant expressions with the following properties:
- No casts are performed.
- Arithmetic is performed using **long int** values.
- The *constant_expression* can contain defined macros. No other identifiers can appear in the expression.
- The *constant_expression* can contain the unary operator **defined**. This operator can be used only with the preprocessor keyword **#if** or **#elif**. The following expressions evaluate to `1` if the *identifier* is defined in the preprocessor, otherwise to `0`:

  ```
  defined identifier
  defined(identifier)
  ```

  For example:

  ```
  #if defined(TEST1) || defined(TEST2)
  ```

Note: If a macro is not defined, a value of `0` (zero) is assigned to it. In the following example, `TEST` must be a macro identifier:

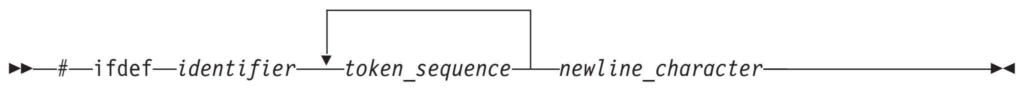
```
#if TEST >= 1
   printf("i = %d\n", i);
   printf("array[i] = %d\n", array[i]);
#elif TEST < 0
   printf("array subscript out of bounds \n");
#endif
```

#ifdef

The **#ifdef** directive checks for the existence of macro definitions.

If the identifier specified is defined as a macro, the lines of code that immediately follow the condition are passed on to the compiler.

The preprocessor **#ifdef** directive has the form:



The following example defines `MAX_LEN` to be `75` if `EXTENDED` is defined for the preprocessor. Otherwise, `MAX_LEN` is defined to be `50`.

```
#ifdef EXTENDED
#   define MAX_LEN 75
#else
#   define MAX_LEN 50
#endif
```

#ifndef

The **#ifndef** directive checks whether a macro is not defined.

If the identifier specified is not defined as a macro, the lines of code immediately follow the condition are passed on to the compiler.

The preprocessor **#ifndef** directive has the form:

```
                             ┌──────────────────┐
                             ▼                  │
►►──#──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
```

#else

If the condition specified in the **#if**, **#ifdef**, or **#ifndef** directive evaluates to `0`, and the conditional compilation directive contains a preprocessor **#else** directive, the lines of code located between the preprocessor **#else** directive and the preprocessor **#endif** directive is selected by the preprocessor to be passed on to the compiler.

The preprocessor **#else** directive has the form:

```
              ┌──────────────────┐
              ▼                  │
►►──#──else──────token_sequence──┴──newline_character─────────────────────────►◄
```

#endif

The preprocessor **#endif** directive ends the conditional compilation directive.

It has the form:

```
►►──#──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);
}
```

Line Control (#line)

A *preprocessor line control directive* supplies line numbers for compiler messages. It causes the compiler to view the line number of the next source line as the specified number.

A preprocessor **#line** directive has the form:

```
►►──#──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.

AIX 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.

Example of the #line Directive

You can use **#line** control directives to make the compiler provide more meaningful error messages. The following program uses **#line** control directives to give each function an easily recognizable line number:

```
/**
 ** This example illustrates #line directives.
 **/

#include <stdio.h>
#define LINE200 200

int main(void)
{
   func_1();
   func_2();
}

#line 100
func_1()
{
   printf("Func_1 - the current line number is %d\n",_ _LINE_ _);
}

#line LINE200
func_2()
{
   printf("Func_2 - the current line number is %d\n",_ _LINE_ _);
}
```

This program produces the following output:

```
Func_1 - the current line number is 102
Func_2 - the current line number is 202
```

Null Directive (#)

The *null directive* performs no action. It consists of a single # on a line of its own.

The null directive should not be confused with the # operator or the character that starts a preprocessor directive.

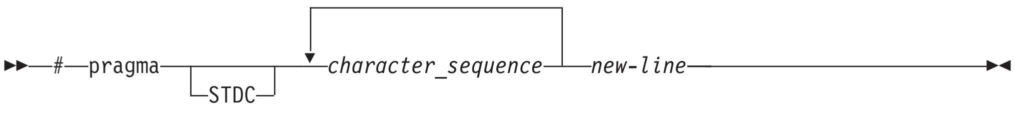
In the following example, if MINVAL is a defined macro name, no action is performed. If MINVAL is not a defined identifier, it is defined 1.

```
#ifdef MINVAL
  #
#else
  #define MINVAL 1
#endif
```

Related References
- "# Operator" on page 165

Pragma Directives (#pragma)

A *pragma* is an implementation-defined instruction to the compiler. It has the general form:



where *character_sequence* is a series of characters giving a specific compiler instruction and arguments, if any. The token STDC indicates a standard pragma; consequently, no macro substitution takes place on the directive. The *new-line* character must terminate a pragma directive.

The *character_sequence* on a pragma is subject to macro substitutions. For example,

```
#define
XX_ISO_DATA
isolated_call(LG_ISO_DATA)
// ...
#pragma XX_ISO_DATA
```

More than one pragma construct can be specified on a single **#pragma** directive. The compiler ignores unrecognized pragmas.

The available pragmas are discussed in *XL C Compiler Reference*.

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─────┬──┬─DEFAULT─┬──new-line──────────────────►◄
                   ├─FENV_ACCESS─────┤  ├─ON──────┤
                   └─CX_LIMITED_RANGE┘  └─OFF─────┘
```

The default for `#pragma STDC CX_LIMITED_RANGE` is `OFF`.

The C standard pragmas are discussed in *XL C Compiler Reference*.

The _Pragma Operator

The unary operator `_Pragma` allows a preprocessor macro to be contained in a pragma directive. A `_Pragma` expression has the following form:

```
►►──_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 ( "align(power)" )
```

would be equivalent to

```
#pragma align(power)
```

Appendix A. The IBM C Language Extensions

This appendix presents the IBM C extensions by category. The major categories are whether an extension is orthogonal or non-orthogonal to a base language. An orthogonal extension does not interfere with the base language. Orthogonal extensions are collectively enabled by compiling in one of the extended modes: `extended`, `extc89`, and `extc99`. The `extended` mode is based on C89.

Non-orthogonal extensions on the other hand may change the syntax or semantics of a base language feature. Therefore, each IBM C extension that is non-orthogonal to the base language or that conflicts with its GNU C implementation must be explicitly requested by an option.

The syntax for the positive and negative `langlvl` suboptions is:

```
-qlanglvl=lang_suboption
-qlanglvl=nolang_suboption
```

Options and suboptions are case-insensitive.

Orthogonal Extensions

The orthogonal IBM C extensions fall into three subgroupings: language features with individual option controls from previous releases, those that are C99 features, and those related to GNU C.

Existing IBM C Extensions with Individual Option Controls

Some existing language features that are orthogonal to C89 have individual positive and negative option controls. For backward compatibility, these compiler options and suboptions continue to be supported. Enabling a feature redundantly will not change its enabled state.

The IBM C language extensions with individual option controls

Language Extension	Compiler Option	Remarks
dollar sign in identifier	`-qdollar`	Accepted by all levels.
UCS	`-qlanglvl=ucs`	The negative setting, `-qlanglvl=noucs`, is ignored by STDC99 with an informational message.
digraph	`-qdigraph`	The negative setting, `-qnodigraph`, is ignored by STDC99 with an informational message.

IBM C Extensions: C99 Features as Extensions to C89

Most of the language features related to C99 are orthogonal to C89. The exception is the **restrict** keyword, which invades the user's variable name space. You can request the support explicitly by using the `-qkeyword=restrict` option.

C99 features as extensions to C89

Language Feature	Remarks
The **restrict** type qualifier	Defines a restricted pointer

C99 features as extensions to C89

Language Feature	**Remarks**
Variable length arrays	`-qlanglvl=c99vla`
Flexible array members	C99 allows a flexible array member only at the end of a struct. GNU C allows it anywhere in the structure.
Support for the complex data type	
The `long long int` type	
Support for hexadecimal floating-point constants	
Removal of implicit `int`	
Refined definition of integer division	Truncation toward zero
Universal character names	
Extended identifiers	Limit removed for internal and external names
Compound literals	
Designated initializers	
C++ style comments	
Removal of implicit function declaration	
Preprocessor arithmetic done in `intmax_t/uintmax_t`	
Mixed declarations and code	
New block scopes for selection and iteration statements	
Integer constant type rules	To accommodate the `long long int` type
Integer promotion rules	To accommodate the `long long int` type
vararg macros	Function-like macros with variable arguments
Trailing comma allowed in `enum` declaration	
Definition of the `_Bool` type	
Idempotent type qualifiers	Also known as ″duplicate type qualifiers″
Empty macro arguments	
Additional predefined macro names	
_Pragma preprocessing operator	
Standard pragmas	`#pragma STDC FP_CONTRACT#pragma STDC FENV_ACCESS#pragma STDC CX_LIMITED_RANGE`
`__func__` predefined identifier	
UTF-16, UTF-32 literals	

IBM C Extensions Related to GNU C

The IBM C compiler recognizes the following subset of the GNU C language extensions. The descriptive labels used in the following table are similar to those in the GNU C documentation.

The IBM C extensions related to GNU C

Language Feature	**Remarks**
Statements and Declarations in Expressions	
Locally Declared Labels	
Labels as Values	Including computed **goto** statements
Nested Functions	
Referring to a Type with **typeof**	The alternate spelling, **__typeof__**, is recommended.
Generalized Lvalues	
Double-Word Integers	
GNU C Complex Types	
GNU C Hexadecimal Float Constants	
Arrays of Length Zero	
Arrays of Variable Length	
Macros with a Variable Number of Arguments	Using an identifier in place of __VA_ARGS__
Non-Lvalue Arrays May Have Subscripts	
Non-Constant Initializers	
Compound Literals	
Cast to a Union Type	
Declaring Attributes of Functions	
Function prototype overriding a nonprototype definition	
`__alignof__` to inquire about the alignment	
Specifying Attributes of Variables	
Specifying Attributes of Types	
Assembler Instructions with C Expression Operands	
Variables in Specified Registers	The compiler accepts the GNU syntax, but ignores the semantics.
Alternate Keywords	
`#warning`	
`#include_next`	

Non-Orthogonal Extensions

The non-orthogonal IBM C extensions fall into three subgroupings: language features from previous releases, those that are C99 features, and those related to GNU C.

Existing IBM C Extensions with Individual Option Controls

Strictly speaking, the IBM C language feature `upconv` is correctly classified as non-orthogonal. However, it is automatically enabled as part of the `extended` language level. This is the major difference between `extended` and `extc89`.

The non-orthogonal IBM C language extensions

Language Extension	Compiler Option	Remarks
`long long` literal	`-qlonglit`	Ignored by stdc99 with a warning.
`upconv`	`-qupconv`	Available by default at the `-qlanglvl=extended` language level.

IBM C Extensions: C99 Features as Extensions to C89

The non-orthogonal IBM C language extensions

Language Extension	Remarks
The `inline` keyword	Non-orthogonal to C89 and GNU C.
Flexible array members	C99 allows a flexible array member only at the end of a struct. GNU C allows it anywhere in the structure.

IBM C Extensions Related to GNU C

The non-orthogonal GNU C extension

Language Extension	Compiler Suboption and Remarks
Macros with a Variable Number of Arguments	Removing the trailing comma when no variable arguments are specified.

Appendix B. Predefined Macros Related to Language Features

The predefined macros provided for XL C fall into two general categories: those related to language features and those related to the AIX platform. Those related to language features are presented here. The platform-related macros are described in *XL C Compiler Reference*.

The following macros test or enable C99 features, features related to GNU C/C++, and other IBM language extensions. A macro is defined to value of 1 if the listed feature is supported under the specified `qlanglvl` suboption. If the feature is not supported, then the macro is undefined. All predefined macros are protected.

Predefined Macros for C99 Features, Features Related to GNU C/C++ and Other IBM Extensions

Feature	**Predefined Macro Name**	**Supported in -qlanglvl Suboption**
Complex number	_COMPLEX_I	Requires an appropriate suboption of compiler option `-qlanglvl`.
Nested function	_IBM_NESTED_FUNCTION	extc89, extc99, extended
flexible array member	__C99_FLEXIBLE_ARRAY_MEMBER	stdc99, extc99
duplicated type qualifier	__C99_DUP_TYPE_QUALIFIER	stdc99, extc99, extc89, extended
new limit for #line	__C99_MAX_LINE_NUMBER	stdc99, extc99, extc89, extended
_Bool type	__C99_BOOL	stdc99, extc99, extc89, extended
long long type	__C99_LLONG	stdc99, extc99
inline function specifier	__C99_INLINE	stdc99, extc99, extc89, extended
restrict qualifier	__C99_RESTRICT	stdc99, extc99 -qkeyword=restrict
static keyword in array declaration	__C99_STATIC_ARRAY_SIZE	stdc99, extc99, extc89, extended
universal character name	__C99_UCN	stdc99, extc99, extc89, extended
variable length arrays	__C99_VAR_LEN_ARRAY	stdc99, extc99, extc89, extended
__func__ keyword	__C99__FUNC__	stdc99, extc99, extc89, extended
hexadecimal floating constants	__C99_HEX_FLOAT_CONST	stdc99, extc99, extc89, extended
C++ style comments	__C99_CPLUSCMT	stdc99, extc99
compound literals	__C99_COMPOUND_LITERAL	stdc99, extc99, extc89, extended
designated initialization	__C99_DESIGNATED_INITIALIZER	stdc99, extc99, extc89, extended
mixed declaration and code	__C99_MIXED_DECL_AND_CODE	stdc99, extc99, extc89, extended
function-like macros with variable arguments	__C99_MACRO_WITH_VA_ARGS	stdc99, extc99, extc89, extended
empty macro arguments	__C99_EMPTY_MACRO_ARGUMENTS	stdc99, extc99, extc89, extended
standard pragmas	__C99_STD_PRAGMAS	stdc99, extc99, extc89, extended
_Pragma operator	__C99_PRAGMA_OPERATOR	stdc99, extc99, extc89, extended
complex type	__C99_COMPLEX	stdc99, extc99, extc89, extended
type generic macros <tgmath.h>	__C99_TGMATH	stdc99, extc99, extc89, extended
implicit function declaration not supported	__C99_REQUIRE_FUNC_DECL	stdc99
concatenation of wide string and non-wide string	__C99_MIXED_STRING_CONCAT	stdc99, extc99, extc89, extended
subscripting in non-lvalue arrays	__C99_NON_LVALUE_ARRAY_SUB	stdc99, extc99, extc89, extended
non-constant array initializers	__C99_NON_CONST_AGGR_INITIALIZER	stdc99, extc99, extc89, extended
GNU C inline asm	__IBM_GCC_ASM	extc89, extc99, extended
local labels	__IBM_LOCAL_LABEL	extc99, extc89, extended
__alignof__	__IBM__ALIGNOF__	extc99, extc89, extended
__typeof__ keyword	__IBM__TYPEOF__	extc99, extc89, extended, -qkeyword=typeof
typeof keyword	__IBM__TYPEOF__	-qkeyword=typeof
function attributes	__IBM_ATTRIBUTES	extc99, extc89, extended

Predefined Macros for C99 Features, Features Related to GNU C/C++ and Other IBM Extensions

Feature	Predefined Macro Name	Supported in -qlanglvl Suboption
type attributes	__IBM_ATTRIBUTES	extc99, extc89, extended
variable attributes	__IBM_ATTRIBUTES	extc99, extc89, extended
dollar signs in identifiers	__IBM_DOLLAR_IN_ID	extc99, extc89, extended
generalized lvalues	__IBM_GENERALIZED_LVALUE	extc99, extc89, extended
gnu 89 __inline__ support	__IBM_GCC__INLINE__	extc99, extc89, extended
explicit register variables	__IBM_REGISTER_VARS	extc99, extc89, extended
alternate keywords	__IBM_ALTERNATE_KEYWORDS	extc99, extc89, extended
__extension__	__IBM_EXTENSION_KEYWORD	extc99, extc89, extended
#assert, #unassert, #cpu, #machine, #system	__IBM_PP_PREDICATE	extc99, extc89, extended
#warning	__IBM_PP_WARNING	extc99, extc89, extended
#include_next	__IBM_INCLUDE_NEXT	extc99, extc89, extended
UTF-16 and UTF-32 string literals	__IBM_UTF_LITERALS	extc99, extc89, extended

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.

Programming Interface Information

Programming interface information is intended to help you create application software using this program.

General-use programming interfaces allow the customer to write application software that obtains the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification, and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and Service Marks

Industry Standards

The following standards are supported:

- The C language is consistent with the International Standard C (ANSI/ISO-IEC 9899–1990 [1992]). This standard has officially replaced American National Standard for Information Systems-Programming Language C (X3.159–1989) and is technically equivalent to the ANSI C standard. The compiler supports the changes adopted into the C Standard by ISO/IEC 9899:1990/Amendment 1:1994.
- The C language is consistent with the International Standard for Information Systems-Programming Language C (ISO/IEC 9899–1999 (E)).

Index

Program Number: 5724-I10

SC09-7895-00