Microsoft® C Optimizing Compiler

for the MS-DOS® Operating System

Language Reference

Microsoft Corporation

If you have comments about the software, complete the Software Problem Report at the back of this manual and return it to Microsoft Corporation.

If you have comments about the software documentation, complete the Documentation Feedback reply card at the back of this manual and return it to Microsoft Corporation.

Document No. 510830022-100-O00-0587

Contents

Tables

Chapter 1
Introduction

1.1 Overview of the C Language

The C language is a general-purpose programming language known for its efficiency, economy, and portability. While these characteristics make it a good choice for almost any kind of programming, C has proven especially useful in systems programming because it facilitates writing fast, compact programs that are readily adaptable to other systems. Well-written C programs are often as fast as assembly-language programs, and they are typically easier to read and maintain.

C was designed to combine efficiency and power in a relatively small language. C does not include built-in functions to perform tasks such as input and output, storage allocation, screen manipulation, and process control. C programmers rely on run-time libraries to perform such tasks.

This design makes C both flexible and compact. Because the language is relatively sparse, it neither assumes nor imposes a particular programming model. You can use the run-time routines supplied, or tailor your own variations for special purposes. The design also helps to isolate language features from processor-specific features in a particular C implementation, which makes it easier to write portable code. While the strict definition of the language makes it independent of any particular operating system or machine, you can easily add system-specific routines to take advantage of the most efficient features of a particular machine.

Note

Microsoft is committed to conformance with the developing standard for the C language as set forth in the Draft Proposed American National Standard—Programming Language C (herinafter referred to as the ANSI C standard). Microsoft extensions to the ANSI C standard are noted in the remaining text. Because the extensions are not a part of the ANSI C standard, their use may restrict portability of programs between systems. See your User's Guide for information on enabling and disabling Microsoft extensions.

The C language includes the following significant features:

- C provides a full set of loop, conditional, and transfer statements to control program flow logically and efficiently and to encourage structured programming.
- C offers an unusually large set of operators. Many of these operators correspond to common machine instructions, allowing a direct translation into machine code. The variety of operators allows you to specify different kinds of operations clearly and with a minimum of code.
- C data types include several sizes of integers, as well as single- and double-precision floating-point types. You can also design more complex data types, such as arrays and data structures, to suit specific program needs.
- C allows you to declare "pointers" to variables and functions. A pointer to an item corresponds to the item's machine address. You can use pointers to make programs more efficient, since pointers let you refer to items in the same way the machine does. C also supports pointer arithmetic, which allows you to access and manipulate memory addresses directly.
- The C preprocessor acts on the text of files before they are compiled. You can use the C preprocessor to define program constants, substitute fast macro definitions for function calls, and compile parts of programs based on specified conditions. The preprocessor is not limited to processing C files; you can use it with any text file.
- C is a flexible language, which leaves many programming decisions up to you. In keeping with this attitude, C imposes few restrictions in matters such as type conversion. Although this characteristic of the language can make your programming job easier, you must know the language well to understand how programs will behave.

1.2 About This Manual

The *Microsoft C Optimizing Compiler Language Reference* defines the C language as implemented by Microsoft Corporation. It is intended as a reference for programmers experienced in C or in another programming language. Thorough knowledge of programming fundamentals is assumed.

Note

Appendix A of this manual provides a quick comparison between Microsoft C and the definition of C found in Appendix A of *The C Programming Language*, by Brian W. Kernighan and Dennis M. Ritchie. Appendix B of this manual summarizes the syntax of the C language as defined by Microsoft.

The run-time library functions available for use in Microsoft C programs are discussed in a separate manual, the *Microsoft C Optimizing Compiler Run-Time Library Reference* .

Consult your *Microsoft C Optimizing Compiler User's Guide* for an explanation of how to compile and link C programs on your system. The *User's Guide* also contains information specific to the implementation of C on your system.

This manual is organized as follows:

Chapter 2, "Elements of C," describes the letters, numbers, and symbols that can be used in C programs and the combinations of characters that have special meanings to the C compiler.

Chapter 3, "Program Structure," discusses the components and structure of C programs and explains how C source files are organized.

Chapter 4, "Declarations," describes how to specify the attributes of C variables, functions, and user-defined types. C provides a number of predefined data types and allows the programmer to declare "aggregate" types and pointers.

Chapter 5, "Expressions and Assignments," describes the operands and operators that form C expressions and assignments. The chapter also discusses the type conversions and side effects that may occur when expressions are evaluated.

Chapter 6, "Statements," describes C statements, which control the flow of program execution.

Chapter 7, "Functions," discusses C functions. In particular, this chapter explains how to define, declare, and call functions and describes function parameters and return values.

Chapter 8, "Preprocessor Directives and Pragmas," describes the instructions recognized by the C preprocessor, a text processor that is automatically invoked before compilation. This chapter also introduces "pragmas," (special instructions to the compiler that you may place source files).

Appendix A, "Differences," lists the differences between Microsoft C and the description of the C language found in Appendix A of *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie.

Appendix B, "Syntax Summary," summarizes the syntax of the C language as implemented by Microsoft.

The remainder of this chapter describes the notational conventions used throughout the manual.

1.3 Notational Conventions

This manual uses the following notational conventions:

Convention	Meaning
Bold	Bold type indicates text that must be typed exactly as shown. Text that is shown in bold type includes C keywords, such as **goto** and **char**, and operators, such as the addition operator (**+**) and the multiplication operator (*****).
Italics	Terms in italics mark the places in syntax descriptions and in the text where specific terms appear in an actual C program. For example, in **goto** *name*; *name* appears in italics to show that this is a general form for the **goto** statement. In an actual program statement, you must supply a particular identifier for the placeholder *name*. Occasionally, italics are used to emphasize particular words in the text.
`Examples`	Examples of C programs and program elements appear in a special typeface to look similar to

listings on the screen or the output of commonly used computer printers:

```
int x, y;
.
.
.
swap (&x, &y);
```

Ellipsis dots

.
.
.

Ellipsis dots may be vertical or horizontal. In the following example, the vertical ellipsis dots indicate that zero or more declarations, followed by one or more statements, may appear between the braces:

{
⟦*declaration*⟧
.
.
.
statement
⟦*statement*⟧
.
.
.
}

Vertical ellipsis dots are also used in program examples to indicate that a portion of the program has been omitted. For instance, in the following excerpt, two program lines are shown. The ellipsis dots between the lines indicate that additional program lines appear between these two lines but are not shown:

```
int x, y;
.
.
.
swap (&x, &y);
```

Horizontal ellipsis dots following an item indicate that more items of the same form may appear. For instance,

= { *expression* ⟦, *expression*⟧...}

indicates that one or more expressions separated

	by commas may appear between the braces ({ }).
⟦Double brackets⟧	Double brackets enclose optional items in syntax descriptions. For example, **return** ⟦*expression*⟧; is a syntax description showing that *expression* is an optional item in the **return** statement.
"Quotation marks"	Quotation marks set off terms defined in the text. For example, the term "token" appears in quotation marks when it is defined. Some C constructs, such as strings, require quotation marks. Quotation marks required by the language have the form `""` rather than " ". For example, `"abc"` is a C string.
SMALL CAPITALS	Names of special key combinations, such as CONTROL-Z, appear in small capital letters.

Chapter 2
Elements of C

2.1 Introduction

This chapter describes the elements of the C programming language, including the names, numbers, and characters used to construct a C program. The following topics are discussed in the remainder of this chapter.

- Character sets
- Constants
- Identifiers
- Keywords
- Comments
- Tokens

2.2 Character Sets

Two character sets are defined for use in C programs: the "C character set" and the "representable character set."

The C character set consists of the letters, digits, and punctuation marks having specific meanings in the C language. You construct a C program by combining the characters of the C character set into meaningful statements.

The C character set is a subset of the representable character set. The representable character set includes each letter, digit, and symbol that can be represented graphically with a single character. The extent of the representable character set depends on the type of terminal, console, or character device being used.

In general, all characters in a C program must be part of the C character set. However, string literals, character constants and comments can include any character from the representable character set.

Since each character in the C character set has an explicit meaning in the language, the compiler generates error messages when it finds inappropriate or inappropriately used characters in a program.

Sections 2.2.1 – 2.2.5 describe the characters and symbols of the C character set and explain how and when to use them.

2.2.1 Letters, Digits, and Underscore

The C character set includes the uppercase and lowercase letters of the English alphabet, the 10 decimal digits of the Arabic number system, and the "underscore" character:

- Uppercase English letters

  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
- Lowercase English letters

  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
- Decimal digits

  0 1 2 3 4 5 6 7 8 9
- Underscore character (_)

These letters and digits are used to form the constants, identifiers, and keywords described later in this chapter.

The C compiler treats uppercase and lowercase letters as distinct characters. For example, if a lowercase a is specified, you cannot substitute an uppercase A ; you must use the lowercase letter.

2.2.2 White-Space Characters

Space, tab, line-feed, carriage-return, form-feed, vertical-tab, and new-line characters are called "white-space characters" because they serve the same purpose as the spaces between words and lines on a printed page. These characters separate the items you define, such as constants and identifiers, from other items in a program.

The C compiler treats a CONTROL-Z character as an end-of-file indicator. It ignores any text after the CONTROL-Z mark.

The C compiler ignores white-space characters unless you use them as separators or as components of character constants or string literals. Therefore, you can use extra white-space characters to make a program more readable. The compiler also treats comments as white space. (Comments are described in Section 2.6.)

2.2.3 Punctuation and Special Characters

The punctuation and special characters in the C character set have various uses, from organizing program text to defining the tasks that the compiler or compiled program will carry out. Table 2.1 lists the punctuation and special characters in the C character set.

Table 2.1

Punctuation and Special Characters

Character	Name	Character	Name
,	Comma	!	Exclamation mark
.	Period	¦	Vertical bar
;	Semicolon	/	Forward slash
:	Colon	\	Backslash
?	Question mark	~	Tilde
'	Single quotation mark	+	Plus Sign
"	Double quotation mark	#	Number sign
(	Left parenthesis	%	Percent sign
)	Right parenthesis	&	Ampersand
[	Left bracket	^	Caret
]	Right bracket	*	Asterisk
{	Left brace	–	Minus sign
}	Right brace	=	Equal sign
<	Left angle bracket	>	Right angle bracket

These characters have special meanings in C. Their uses are described throughout this manual. If a punctuation character from the representable character set does not appear in Table 2.1, you can use that character only in string literals, character constants, and comments.

2.2.4 Escape Sequences

Strings and character constants can contain “escape sequences.” Escape sequences are character combinations representing white-space and nongraphic characters. An escape sequence consists of a backslash (\) followed by a letter or combination of digits.

Escape sequences are typically used to specify actions such as carriage returns and tab movements on terminals and printers and to provide literal representations of nonprinting characters and characters that normally have special meanings, such as the double quote (") character. Table 2.2 lists the C escape sequences.

Table 2.2

Escape Sequences

Escape Sequence	**Name**
n	New line
t	Horizontal tab
v	Vertical tab
b	Backspace
r	Carriage return
f	Form feed
a	Bell (alert)
'	Single quote
"	Double quote
\\\\	Backslash
\ddd	ASCII character in octal notation
\**x***ddd*	ASCII character in hexadecimal notation

If a backslash precedes a character that does not appear in Table 2.2, the backslash is ignored and the character is represented literally. For example, the pattern \c represents the character c in a string literal or character constant. However, the use of lower-case letters in escape sequences is reserved by ANSI for future standardization. Therefore, occurrences of undefined escape sequences, though currently innocuous, could pose future portability problems.

The sequence *ddd* allows you to specify any character in the ASCII (American Standard Code for Information Interchange) character set as a three-digit octal character code. Similarly, the sequence **\x***ddd* allows you to specify any ASCII character as a three-digit hexadecimal character code. For example, you can give the ASCII backspace character as `\010` (octal) or `\x008` (hexadecimal)

You can use only the digits 0 through 7 in an octal escape sequence. You must use at least one digit, but you can use fewer than three digits. For example, you can specify the ASCII backspace character in octal notation as `\10`. You must use at least one digit for a hexadecimal escape sequence, but you can omit the second and/or third digits. Therefore you could specify the hexadecimal escape sequence for the backspace character either as `\x08` or as `\x8`.

Note

When you use octal and hexadecimal escape sequences in strings, it is safest to give all three digits of the escape sequence. If you don't specify all digits of the escape sequence, and the character immediately following the escape sequence happens to be an octal or hexadecimal digit, the compiler interprets that character as part of the sequence. For example, if you printed the string `"\x07Bell"`, the result would be `{ell` because `\x07B` is interpreted as the ASCII left-brace character (`{`). The string `\x007Bell` (note the two leading zeros) is the correct way to represent the bell character followed by the word `Bell`. The string `\x7Bell` would generate a compiler diagnostic message because 7beH is too big a number to fit in one byte.

Escape sequences allow you to send nongraphic control characters to a display device. For example, the escape character, `\033`, is often used as the first character of a control command for a terminal or printer. Some escape sequences are device specific. For instance, the vertical tab and form feed (`\v` and `\f`) do not affect screen output, but perform appropriate operations for a printer.

Note

You should always represent nongraphic characters by escape sequences in C programs, since using the characters directly may generate compiler diagnostic messages.

You can also use the backslash character (\) as a continuation character. When a new-line character follows the backslash, the compiler ignores the backslash and the new line and treats the next line as part of the previous line. This is useful primarily for preprocessor definitions longer than a single line. In the past this feature was also used to create strings longer than one line. However the string concatenation feature (see Section 2.3.4) is preferred for creating long string literals.

2.2.5 Operators

Operators are symbols (both single characters and character combinations) that specify how values are to be manipulated. Each symbol is interpreted as a single unit, called a "token." (Tokens are defined in Section 2.7.)

Table 2.3 lists the symbols comprising the C unary operators and names each operator. Table 2.4 lists the C binary and ternary operators and names them. You must specify operators exactly as they appear in the tables, with no white space between the characters of multicharacter operators. Note that three operator symbols (asterisk, minus sign, and ampersand) appear in both tables. Their interpretation as unary or binary depends on the context in which they appear. The **sizeof** operator is not included in these tables. It consists of a keyword (**sizeof**) rather than a symbol, and is listed in Section 2.5.

Table 2.3

Unary Operators

Operator	Name
!	Logical NOT
~	Bitwise complement
–	Arithmetic negation
*	Indirection
&	Address of

Table 2.4
Binary and Ternary Operators

Operator	Name
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder
<<	Left shift
>>	Right shift
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
==	Equality
!=	Inequality
&	Bitwise AND
¦	Bitwise inclusive OR
^	Bitwise exclusive OR
&&	Logical AND
¦¦	Logical OR
,	Sequential evaluation
?:	Conditional[a]
++	Increment
--	Decrement
=	Simple assignment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
=	Remainder assignment
>>=	Right-shift assignment
<<=	Left-shift assignment
&=	Bitwise AND assignment

=	Bitwise inclusive OR assignment
^=	Bitwise exclusive OR assignment

[a] The conditional operator is a ternary operator, not a multicharacter operator. A conditional expression has the following form: *expression* **?** *expression* **:** *expression*

For a complete description of each operator, see Chapter 5, "Expressions and Assignments."

2.3 Constants

A constant is a number, character, or character string that can be used as a value in a program. A constant's value may not be modified by the program in which it occurs.

The C language has four kinds of constants: integer constants, floating-point constants, character constants, and string literals. Sections 2.3.1 – 2.3.4 describe the format and use of each kind of constant.

2.3.1 Integer Constants

Syntax

digits

0*odigits*

0x*hdigits*
0X*hdigits*

An integer constant is a decimal, octal, or hexadecimal number that represents an integer value.

- A decimal constant has the form *digits*, where *digits* represents one or more decimal digits (0 through 9).
- An octal constant has the form **0***odigits*, where *odigits* represents one or more octal digits (0 through 7). The leading zero is required.

- A hexadecimal constant has the form **0x***hdigits* or **0X***hdigits*, where *hdigits* represents one or more hexadecimal digits (0 through 9 and either uppercase or lowercase "a" through "f"). The leading zero is required and must be followed by **x** or **X.**

No white-space characters can separate the digits of an integer constant.

Table 2.5 gives examples of the three forms of integer constants.

Table 2.5

Examples of Integer Constants

Decimal Constants	**Octal Constants**	**Hexadecimal Constants**
`10`	`012`	`0xa or 0xA`
`132`	`0204`	`0x84`
`32179`	`076663`	`0x7dB3 or 0x7DB3`

Integer constants always specify positive values. If you need to use a negative value, place a minus sign(–) in front of a constant to form a constant expression with a negative value. (In this case, the minus sign is interpreted as the unary arithmetic negation operator.)

Every integer constant is given a type based on its value. A constant's type determines which conversions must be performed when the constant is used in an expression or when the minus sign (–) is applied:

- Decimal constants are considered signed quantities and are given **int** type, or **long** type if the size of the value requires it.
- Octal and hexadecimal constants are given **int**, **unsigned int**, **long**, or **unsigned long** type, depending on the size of the constant. If the constant can be represented as an **int**, it is given **int** type. If it is larger than the maximum positive value that can be represented by an **int**, but small enough to be represented in the same number of bits as an **int**, it is given **unsigned int** type. Similarly, a constant that is too large to be represented as an **unsigned int** is given **long** or **unsigned long** type, if necessary.

Table 2.6 shows the ranges of values and the corresponding types for octal and hexadecimal constants on a machine where the **int** type is 16 bits long.

Table 2.6

Types Assigned to Octal and Hexadecimal Constants

Hexadecimal Range	Octal Range	Type
0x0 – 0x7FFF	0 – 077777	**int**
0x8000 – 0xFFFF	0100000 – 0177777	**unsigned int**
0x10000 – 0x7FFFFFFF	0200000 – 017777777777	**long**
0x80000000 – 0xFFFFFFFF	020000000000 – 030000000000	**unsigned long**

The consequence of the typing rules shown in Table 2.6 is that hexadecimal and octal constants are always zero-extended when converted to longer types. (For a discussion of type conversions, see Chapter 5, "Expressions and Assignments.")

You can force any integer constant to be given **long** type by appending the letter "l" or "L" to the end of the constant. Table 2.7 illustrates some forms of **long** integer constants.

Table 2.7

Examples of Long Integer Constants

Decimal Constants	Octal Constants	Hexadecimal Constants
`10L`	`012L`	`0xaL or 0xAL`
`79l`	`0115l`	`0x4fl or 0x4Fl`

Types are described in Chapter 4, "Declarations," and conversions are described in Chapter 5, "Expressions and Assignments."

2.3.2 Floating-Point Constants

Syntax

⟦*digits*⟧⟦**.***digits*⟧⟦**E**|**e**⟦–⟧*digits*⟧

A floating-point constant is a decimal number that represents a signed real number. The value of a signed real number includes an integer portion, a fractional portion, and an exponent. The *digits* are zero or more decimal digits (0 through 9), and **E** (or **e**) is the exponent symbol. You can omit either the digits before the decimal point (the integer portion of the value) or the digits after the decimal point (the fractional portion), but not both. You can leave out the decimal point only if you include an exponent.

The exponent consists of the exponent symbol (**E** or **e**) followed by a constant integer value. The integer value may be negative. No white-space characters can separate the digits or characters of the constant.

Floating-point constants always specify positive values. However, you can place a minus sign (–) in front of the constant to form a constant floating-point expression with a negative value. In this case, the minus sign is treated as an arithmetic operator.

All floating-point constants have type **double**.

Examples

The following examples illustrate some forms of floating-point constants and expressions:

```
15.75
1.575E1
1575e-2
-0.0025
-2.5e-3
25E-4
```

You can omit the integer portion of the floating-point constant, as shown in the following examples:

```
.75
.0075e2
-.125
-.175E-2
```

2.3.3 Character Constants

A character constant is formed by enclosing a single character from the representable character set within single quotation marks (' '). An escape sequence is regarded as a single character and is therefore a valid character constant. Note that escape characters must be represented by escape sequences or diagnostic messages will be generated. The value of a character constant is the numerical representation of the character.

A character constant has the form

'*char*'

where *char* can be any character from the representable character set (including any escape sequence) except a single quotation mark ('), backslash (\), or new-line character. To use a single quotation mark or backslash character as a character constant, precede it with a backslash, as shown in Table 2.8. To represent a new-line character, use the escape sequence **\n**.

Table 2.8

Examples of Character Constants

Constant	Value
' '	Single blank space
'a'	Lowercase a
'?'	Question mark
'\b'	Backspace
'\x1B'	ASCII escape character
'\''	Single quotation mark
'\\'	Backslash

Character constants have type **int**, and are therefore sign-extended in type conversions. (See Section 5.7, "Type Conversions," for more information about type conversions.)

2.3.4 String Literals

Syntax

"*characters*" ⟦"*characters*"...⟧

A string literal is a sequence of characters from the representable character set enclosed in double quotation marks (" "). In a string literal, *characters* is a placeholder for zero or more characters from the representable character set, including any escape sequence, except a double quotation mark ("), backslash (\), or new-line character. Escape characters must be represented by escape sequences, and each escape sequence is considered a single character.

```
"This is a string literal."
```

To force a new line within a string literal, enter the new-line (**n**) escape sequence at the point in the string where you want the line broken, as follows:

```
"Enter a number between 1 and 100\nOr press Return"
```

The traditional way to form string literals that take up more than one line is to type a backslash, then press the RETURN key. The backslash causes the compiler to ignore the new-line character. For example, the string literal

```
"Long strings can be bro\
ken into two or more pieces."
```

is identical to the string

```
"Long strings can be broken into two or more pieces."
```

Two or more string literals separated only by white space will be concatenated into a single string. For example, long strings passed as literals to the **printf** function may now be continued in any column of a succeeding line without affecting their appearance when output, if entered as follows:

```
printf ("This is the first half of the string,"
               " this is the second half") ;
```

As long as each part of the string is enclosed in double quotation marks, they will be concatenated, and output as a single string:

```
This is the first half of the string, this is the second half
```

String concatenation can be used anywhere you might previously have used a backslash followed by a new-line character to enter strings longer than one line. Because ensuing strings can start in any column of the source code without affecting their on-screen representation, strings can be positioned to enhance source-code readability. For example, the following pointer, initialized as two separate string literals separated only by white space, is stored as a single string. When properly referenced, as in the following example, it produces a result identical to the example immediately above:

```
char *string = "This is the first half of the string,"
               " this is the second half" ;

printf("%s" , string) ;
```

To use a double quotation mark or backslash within a string literal, precede it with a backslash, as shown in the following examples:

```
"First\\Second"

"\"Yes, I do,\" she said."
```

Note that an escape sequence (such as \\ or \") within a string literal counts as a single character.

The characters of a string are stored in order at contiguous memory locations. A null character (represented by the **\0** escape sequence) is automatically appended to, and marks the end of, each string literal. Each string in a program is generally considered to be distinct; however, two identical strings are not guaranteed to receive separate storage. Therefore, programs should not attempt to modify string literals during execution.

String literals have type array of **char** (**char** [ ]). This means that a string is an array with elements of type **char**. The number of elements in the array is equal to the number of characters in the string, plus one for the terminating null character. The array contains one element in addition to the number of characters in the string literal, since the null character, stored after the last string character, counts as an array element.

2.4 Identifiers

Syntax

letter|_ ⟦*letter*|*digit*|_ ...⟧

Identifiers are the names you supply for variables, functions, and labels in your program. You create an identifier by specifying it in the declaration of a variable or function. You can then use the identifier in later program statements to refer to the associated item. Although statement labels are a special kind of identifier and have their own naming class, their creation is similar to that of variables and functions. (Declarations are described in Chapter 4, "Declarations." Statement labels are described in Chapter 6, "Statements.")

An identifier is a sequence of one or more letters, digits, or underscores (_) that begins with a letter or underscore. Identifiers can contain any number of characters, but only the first 31 characters are significant to the compiler. (Other programs that read the compiler output, such as the linker, may recognize even fewer characters.)

The C compiler considers uppercase and lowercase letters to be distinct characters. Therefore, you can create distinct identifiers that have the same spelling but different cases for one or more of the letters.

An identifier cannot have the same spelling and case as a keyword of the language. Keywords are described in Section 2.5.

You should not use leading underscores in identifiers you create: identifiers beginning with an underscore can cause conflicts with the names of system routines or variables, and produce errors. Programs containing names beginning with leading underscores are not guaranteed to be portable.

Note

Some linkers may further restrict the number and type of characters for globally visible symbols. (Visibility is defined in Section 3.5, "Lifetime and Visibility.") Also the linker, unlike the compiler, may not distinguish between uppercase and lowercase letters. Consult your linker documentation for information about naming restrictions imposed by the linker.

Examples

The following are examples of identifiers:

```
j
cnt
temp1
top_of_page
skip12
```

Since uppercase and lowercase letters are considered distinct characters, each of the following identifiers is unique:

```
add
ADD
Add
aDD
```

2.5 Keywords

"Keywords" are predefined identifiers that have special meanings to the C compiler. They can be used only as defined. The names of program items cannot have the same spelling and case as a C keyword.

The C language has the following keywords:

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

You cannot redefine keywords. However, you can specify text to be substituted for keywords before compilation by using C preprocessor directives (see Chapter 8, "Preprocessor Directives and Pragmas").

The **volatile** keyword is implemented syntactically, but currently has no semantics associated with it. You should not use **volatile** as a variable name in your programs.

The following identifiers may be keywords in some implementations. See your User's Guide for more information.

cdecl
far
fortran
huge
near
pascal

2.6 Comments

Syntax

/ characters */*

A comment is a sequence of characters that is treated as a single white-space character by the compiler, but is otherwise ignored. In a comment, *characters* can include any combination of characters from the representable character set, including new-line characters, but excluding the "end comment" delimiter (*/). Comments can occupy more than one line, but they cannot be nested.

Comments can appear anywhere a white-space character is allowed. Since the compiler treats a comment as a single white-space character, you cannot include comments within tokens (see Section 2.7 for a definition of "token"). However, since the compiler ignores the characters of the comment, you can include keywords in comments without producing errors.

To suppress compilation of a large portion of a program or a program segment that contains comments, use the **#if** preprocessor directive, rather than the practice of "commenting out" the code (see Section 8.4 of Chapter 8, "Preprocessor Directives and Pragmas").

Examples

The following examples illustrate some comments:

```
/* Comments can separate and document
   lines of a program. */
```

```
/* Comments can contain keywords such as for
   and while. */

/*****************************************
   Comments can occupy several lines.
 *****************************************/
```

Since comments cannot contain nested comments, the following example causes an error:

```
/* You cannot /* nest */ comments */
```

The error occurs because compiler recognizes the first `*/`, after the word `nest`, as the end of the comment. It tries process the remaining text and produces an error when it cannot do so.

2.7 Tokens

In a C source program, the basic element recognized by the compiler is the character group known as a "token." A token is source-program text the compiler will not attempt to further analyze into component. For example, the following program fragment uses the word "elsewhere" as the name of a function. Although **else** is a keyword in C, there is no confusion between the function name token and the C keyword token it contains.

```
main()
{
    int i = 0;
        if (i)
        elsewhere() ;
}
```

However, if you were to type `elsewhere` as `else where` with a space between "else" and "where," the preceding example would elicit a compiler error message noting the lack of a semicolon before the **else** keyword.

The operators, constants, identifiers, and keywords described in this chapter are examples of tokens. Punctuation characters such as brackets ([ ]), braces ({ }), angle brackets (< >), parentheses, and commas are also tokens.

Tokens are delimited by white-space characters and by other tokens, such as operators and punctuation characters. To prevent the compiler from breaking an item down into two or more tokens, white-space characters are not permitted within an identifier, multicharacter operator, or keyword.

When the compiler interprets tokens, it includes as many characters as possible in a single token before moving on to the next token. Because of this behavior, the compiler may not interpret tokens as you intended if they are not properly separated by white space.

Example

Consider the following expression:

```
i+++j
```

In this example, the compiler first makes the longest possible operator (++) from the three plus signs, then processes the remaining plus sign as an addition operator (+). Thus, the expression is interpreted as `(i++) + (j)`, not `(i) + (++j)`. In this and similar cases, use white space and parentheses to avoid ambiguity and insure proper expression evaluation.

Chapter 3
Program Structure

3.1 Introduction

This chapter defines terms used later in this manual to describe the C language, and discusses the structure of C source programs. It gives an overview of features of C that are described in detail in other chapters. The syntax and meaning of declarations and definitions are discussed in Chapter 4, "Declarations," and Chapter 7, "Functions." The C preprocessor and pragmas are described in Chapter 8, "Preprocessor Directives and Pragmas."

3.2 Source Program

A C source program is a collection of any number of directives, pragmas, declarations, definitions, and statements. These constructs are discussed briefly in the following paragraphs. To be compiled by the Microsoft C Optimizing Compiler, each must have the syntax described in this manual, and each can appear in any order in the program (subject to the rules outlined throughout this manual). However, order of appearance does affect how variables and functions can be used in a program. (See Section 3.5 for more information.)

Directives

A "directive" instructs the C preprocessor to perform a specific action on the text of the program before compilation. Directives are described in Chapter 8 of this manual, "Preprocessor Directives and Pragmas."

Pragmas

A "pragma" instructs the compiler to perform a particular action at compile time. Pragmas are described in Chapter 8 of this manual, "Preprocessor Directives and Pragmas."

Declarations

A "declaration" establishes an association between the name and the attributes of a variable, function, or type. A "defining declaration" of a variable establishes the same associations, but also gives the variable an initial value. Because nondefining variable declarations are used to declare a reference to a variable defined elsewhere, they are sometimes referred to as "referencing declarations." A variable declared in such a way that it has global lifetime is initialized to zero if no explicit definition appears in in any source file of the program. (Global lifetime is described in Section 3.5).

Function declarations include the name of the function, its return type, and optionally, it's formal parameters. A function definition includes the same elements plus the function body. Both function and variable declarations may appear inside or outside a function definition. Any declaration within a function definition is said to appear at the "internal level." A declaration outside all function definitions is said to appear at the "external level." (Function definitions are discussed further under "Definitions" below.)

Definitions

A definition creates an instance of a variable or specifies the components of a function.

A variable definition tells the compiler to allocate storage for the declared variable and assigns the variable an initial value.

Variable definitions, like declarations, can appear at the internal level (within a function definition) or at the external level (outside all function definitions). Function definitions always occur at the external level.

A function definition includes the "function body," which is a compound statement containing the declarations and statements comprising the function. (Compound statements are described in Chapter 6, "Statements.") The function definition also gives the name, formal parameters, and return type of the function. A nontrivial program always contains at least one function definition, which defines the action that the program will take.

Function declarations may appear at the internal or external level. However, function definitions may only appear at the external level, that is, outside of all other functions.

Example

The following example illustrates a simple C source program:

```
int x = 1;                    /* Variable definitions */
int y = 2;

extern int printf(char *,...);/* Function declaration */

main ()                       /* Function definition
                                 for main function */
{
        int z;                /* Variable declarations */
        int w;

        z = y + x;            /* Executable statements */
        w = y - x;
        printf("z= %d \nw= %d \n", z, w);
}
```

This source program defines the function named `main` and declares the function named `printf`. The program uses variable definitions to define the variables `x` and `y`; it simply declares the variables `z` and `w`.

3.3 Source Files

A source program can be divided into one or more "source files." A C source file is a text file containing all or part of a C source program. (For example, a source file may contain just a few of the functions that the program needs.) When you compile a program, you must separately compile, and then link, the individual source files comprising the total program. You can also use the **#include** directive to combine separate source files into larger source files before you compile. (See Chapter 8, "Preprocessor Directives and Pragmas," Section 8.3 for information on "include" files.)

A source file can contain any combination of complete directives, pragmas, declarations, and definitions. You cannot split items such as function definitions or large data structures between source files. The last character in a source file must be a new-line character.

A source file need not contain executable statements. For example, you may find it useful to place definitions of variables in one source file and then declare references to these variables in other source files that use them. This technique makes the definitions easy to find and change. For the same reason, manifest constants and macros are often organized into separate include files that may be referenced in source files as required.

Directives in a source file apply only to that source file and its include files. Moreover, each directive applies only to the part of the file that follows the directive. To apply a commmon set of directives to a whole source program, you must include the directives in all source files comprising the program.

Pragmas usually affect a specific region of a source file. The implementation determines the specific compiler action that a pragma defines. (Your User's Guide describes the effects of particular pragmas.)

Example

The following example illustrates a C source program contained in two source files. Once you have compiled these source files, you can link and then execute them as a single program.

The main and max functions are assumed to be in separate files, and execution of the program is assumed to begin with the main function.

```
/*************************************************************
        Source file 1 - main function
 *************************************************************/

#define ONE     1
#define TWO     2
#define THREE   3

extern int max(int a, int b);      /* Function declaration */

main ()                        /* Function definition */
{
        int w = ONE, x = TWO, y = THREE;
        int z = 0;
        z = max(x,y);
        w = max(z,w);
}
```

In source file 1 (above), the max function is declared without being defined. This kind of declaration is known as a "forward declaration." The definition for the main function includes calls to max.

The lines beginning with a number sign (#) are preprocessor directives. These directives tell the preprocessor to replace the identifiers ONE, TWO, and THREE with the corresponding number, everywhere in source file 1. However, the directives do not apply to source file 2 (below), which will be separately compiled and then linked with source file 1.

```
/**************************************************************
        Source file 2 - definition of max function
 **************************************************************/

int max (int a, int b)
  {
        if ( a > b )
                return (a);
        else
                return (b);
  }
```

Source file 2 contains the function definition for max. This definition satisfies the calls to max in source file 1. Note that the definition for max follows the form specified in the Draft Proposed American National Standard—Programming Language C (the ANSI C standard).For more information on this new form and function prototyping, see Chapter 7, "Functions."

3.4 Functions and Program Execution

Every C program has a primary (main) program function, which must be named **main**. The **main** function serves as the starting point for program execution. It usually controls program execution by directing the calls to other functions in the program. A program usually stops executing at the end of the **main** function, although it can terminate at other points in the program for a variety of reasons depending on the execution environment.

The source program usually has more than one function, with each function designed to perform one or more specific tasks. The **main** function can call these functions to perform their respective tasks. When **main** calls another function, it passes execution control to the function, so that execution begins at the first statement in the function. The function returns control when a **return** statement is executed or when the end of the function is reached.

You can declare any function, including **main**, to have parameters. When one function calls another, the called function receives values for its parameters from the calling function. These values are called "arguments." You can declare parameters to the **main** function so that **main** receives values from outside the program. (Most commonly, these arguments are passed from the command line when the program is executed.)

When the **main** function takes parameters, they are traditionally named **argc** and **argv**. The **argc** parameter is declared to hold the total number of arguments passed to **main**. The **argv** parameter is declared as an array of pointers; each element of the array points to a string representation of an argument passed to the **main** function.

Traditionally, if a third parameter is passed to the **main** function, that parameter is named **envp**. C does not require this name, however. It is an extension to the ANSI C standard provided by Microsoft C for compatability with the XENIX® Operating System. The **envp** parameter is a pointer to a table of string values that set up the environment in which the program executes.

The operating system supplies values for the **argc**, **argv**, and **envp** parameters, and the user supplies the actual arguments to the **main** function. The operating system, not the C language, determines the argument-passing convention used on a particular system. For more information, see your User's Guide.

If you declare formal parameters to a function, you must declare them when you define the function. Function definitions are described in more detail in Section 7.2. Function declarations are discussed in Section 4.5.

3.5 Lifetime and Visibility

To understand how a C program works, you must understand the rules that determine how variables and functions can be used in the program. Three concepts are crucial to understanding these rules: the "block" (or compound statement), "lifetime" (sometimes called "extent"), and "visibility" (sometimes called "scope").

Blocks

A block is a sequence of declarations, definitions and statements enclosed within curly braces. There are two types of blocks in C. The compound statement (discussed more fully in Chapter 6, "Statements") is one type of block. The other, the function definition, consists of a compound statement comprising the function body plus the function's associated "header" (the function name, return type, and optional formal parameters). A block may encompass other blocks, with the exception that no block may not contain a function definition. A block within other blocks is said to be "nested" within the encompassing blocks.

Note that, while all compound statements are enclosed within curly braces, not everything enclosed within curly braces constitutes a compound statement. For example, the specification of array, structure or enumeration elements may appear within curly braces, but these are not considered compound statements.

Lifetime

Lifetime is the period, during execution of a program, in which a variable or function exists. All functions in a program exist at all times during its execution.

Lifetime of a variable may be "global" or "local." If its lifetime is global (a "global item"), it has storage and a defined value for the entire duration of a program. An item with a "local" lifetime (a "local item") has storage and a defined value only within the block where the item is defined or declared. A local item is allocated new storage each time the program enters that block, and it loses its storage (and hence its value) when the program exits the block. Global items are frequently referred to as "static," while local items are often called "automatic."

The following rules specify whether a variable has global or local lifetime:

- Variables declared at the external level (that is, outside all blocks in the program) always have global lifetimes.
- Variables declared at the internal level (that is, within a block) usually have local lifetimes. However, you can insure global lifetime for a variable within a block by including the **static** storage class specifier in its declaration. Once declared **static**, the variable will retain its value from one entry of the block to the next. However, it will still be "visible" only within its own block and blocks nested within its own block. (Visibility of objects is discussed below. See

Section 4.6 for a discussion of storage-class specifiers.)

Visibility

An item's "visibility" determines the portions of the program in which it can be referenced by name. An item is "visible" only in portions of a program encompassed by it's "scope," which may be limited (in order of increasing restrictiveness) to the file, function, block or function prototype in which it appears.

In C, only a label name is always confined to function scope. (See Chapter 6, "Statements," for more information on labels and label names). The scope of any other item is determined by the level at which its declaration occurs. An item declared at the external level has file scope and is visible everywhere within the file. If its declaration occurs within a block (including the list of parameter identifiers in a function definition), the item's scope is limited to that block and blocks nested within that block. Formal parameter names declared in the parameter list of a function prototype have scope only from the completion of the parameter declaration to the end of the function declarator.

Note

Note that, although an item with a global lifetime *exists* exists throughout the execution of the source program (for example, an externally declared variable or a local variable declared with the **static** keyword), it may not be visible in all parts of the program.

An item is said to be "globally visible" if it is visible, or if you can use appropriate declarations to make it visible, in all the source files comprising the program. (Visibility between source files, also known as "linkage," is discussed in greater detail in Section 4.6, "Storage Classes.")

The following rules govern the visibility of variables and functions within a program:

- Variables declared or defined at the external level (that is, outside all blocks in the program) are visible from their point of definition or declaration to the end of the source file. You can use appropriate declarations to make such variables visible in other source files, as described in Section 4.6, "Storage Classes." However, variables declared at the external level with the **static** storage-class specifier are visible only within the source file in which they are defined.

- In general, variables declared or defined at the internal level (that is, within a block) are visible only from their point of declaration or definition to the end of the block actually containing the definition or declaration. Such variables are known as a "local" variables.
- Variables from outer blocks (including those declared at the external level) are visible in all inner blocks. However, the visibility of variables is said to "nest" within blocks. For instance, a block within another block can contain declarations for variables whose identifiers (names) are the same as variables in enclosing blocks. Such redefinitions prevail only within the inner block, however. Outer-block definitions are restored as the inner blocks are exited.
- Functions with **static** storage class are visible only in the source file in which they are defined. All other functions are globally visible. (For more information on function declarations, see Section 4.5.)

Summary

Table 3.1 summarizes the main factors determining lifetime and visibility of variables and functions. However, the table does not cover all possible cases. Refer to the previous discussion and to Section 4.6, "Storage Classes," for more information.

Note

A Microsoft extension to the ANSI C standard provides that functions declared at an internal level may have global visibility. This feature should not be relied upon where portability of source code is a consideration. See your User's Guide for more information.

Table 3.1

Summary of Lifetime and Visibility

Level	Item	Storage Class Specifier	Lifetime	Visibility
External	Variable declaration	**static**	Global	Restricted to single source file
	Variable declaration	**extern**	Global	Remainder of source file
	Function declaration or definition	**static**	Global	Restricted to single source file
	Function declaration or definition	**extern**	Global	Remainder of source file
Internal	Variable definition or declaration	**extern** or **static**	Global	Block
	Variable definition or declaration	**auto** or **register**	Local	Block

Example

The following program example illustrates blocks, nesting, and visibility of variables:

```
#include <stdio.h>

/* i defined at external level: */
int i = 1;

/* main function defined at external level: */
main ()
{

    /* prints 1 (value of external level i): */
    printf("%d\n", i);

    /* begin first nested block: */
    {
        /* i and j defined at internal level: */
        int i = 2, j = 3;

        /* prints 2, 3: */
        printf("%d\n%d\n", i, j);

        /* begin second nested block: */
        {

            /* i is redefined: */
            int i = 0;

            /* prints 0, 3: */
            printf("%d\n%d\n", i, j);

        /* end of second nested block: */
        }

        /* prints 2 (outer definition restored): */
        printf("%d\n", i);

    /* end of first nested block: */
    }

    /* prints 1 (external level definition restored): */
    printf("%d\n", i);
}
```

In this example, there are four levels of visibility: the external level and three block levels. Assuming that the function `printf` is defined elsewhere in the program, the values will be printed to the screen as noted in the comments preceding each statement.

3.6 Naming Classes

In any C program, identifiers are used to refer to many different kinds of items. When you write a C program, you provide identifiers for the functions, variables, formal parameters, union members, and other items the program uses. C allows you to use the same identifier for more than one program item, as long as you follow the rules outlined in this section.

The compiler sets up "naming classes" to distinguish between the identifiers used for different kinds of items. The names within each class must be unique to avoid conflict, but an identical name can appear in more than one naming class. This means that you can use the same

identifier for
two or more different items, provided that the items are in different naming classes. The compiler can resolve references based on the context of the identifier in the program.

The following list describes the kinds of items you can name in C programs and the rules for naming them:

Items	**Naming Class**
Variables and functions	The names of variables and functions are in a naming class with formal parameters, typedef names and enumeration constants. Therefore, variable and function names must be distinct from other names in this class that have the same visibility. However, you can redefine variable and function names within program blocks, as described in Section 3.5, "Lifetime and Visibility."
Formal parameters	The names of formal parameters to a function are grouped with the names of the function's variables, so the formal parameter names should be distinct from the variable names. You cannot redeclare the formal parameters at the top level of the function. However, the names of the formal parameters may be redefined (that is used to refer to different items) within subsequent blocks nested within the function body.
Enumeration constants	Enumeration constants are in the same naming class as variable and function names. This means that the names of enumeration constants must be distinct from all variable and function names with the same visibility, and distinct from the names of other enumeration constants with the same visibility. However, like variable names, the names of enumeration constants have nested visibility, so you can redefine them within blocks. (Nested visibility is discussed in

Section 3.5, “Lifetime and Visibility.”)

typedef names	The names of types defined with the **typedef** keyword are in a naming class with variable and function names. Therefore, **typedef** names must be distinct from all variable and function names with the same visibility, and also from the names of formal parameters and enumeration constants. Like variable names, names used for **typedef** types can be redefined within program blocks. See Section 3.5, “Lifetime and Visibility.”
Tags	Enumeration, structure, and union tags are grouped in a single naming class. Each enumeration, structure, or union tag must be distinct from other tags with the same visibility. Tags do not conflict with any other names.
Members	The members of each structure and union form a naming class. The name of a member must, therefore, be unique within the structure or union, but it does not have to be distinct from other names in the program, including the names of members of different structures and unions.
Statement labels	Statement labels form a separate naming class. Each statement label must be distinct from all other statement labels in the same function. Statement labels do not have to be distinct from other names or from label names in other functions.

Example

```
struct student {
        char student[20];
        int class;
        int id;
        } student;
```

Since structure tags, structure members, and variable names are in three different naming classes, the three items named `student` in this example do not conflict. The context of each item allows the compiler to correctly interpret each occurrence of `student` in the program.

For example, when `student` appears after the **struct** keyword, the compiler recognizes it as a structure tag. When `student` appears after a member-selection operator (**->** or **.**), the name refers to the structure member. In other contexts, `student` refers to the structure variable.

Chapter 4
Declarations

4.1 Introduction

This chapter describes the form and constituents of C declarations for variables, functions, and types. C declarations have the form

⟦*sc-specifier*⟧ ⟦*type-specifier*⟧ *declarator*⟦= *initializer*⟧ ⟦,*declarator*⟦= *initializer*⟧...⟧

where *sc-specifier* is a storage-class specifier; *type-specifier* is the name of a defined type; *declarator* is an identifier; and *initializer* gives the value or sequence of values to be assigned to the variable being declared.

You must explicitly declare all C variables before using them. You can declare a C function explicitly by declaring it, or implicitly by calling the function before you define or declare it.

The C language includes a standard set of data types. You can add your own data types by declaring new ones based on types already defined. You can declare arrays, data structures, and pointers to both variables and functions.

C declarations require one or more *declarators*. A declarator is an identifier that can be modified with brackets ([ ]), asterisks (*), or parentheses (( )) to declare an array, pointer, or function type, respectively. When you declare simple variables (such as character, integer, and floating-point items), or structures and unions of simple variables, the declarator is just an identifier.

Four storage-class specifiers are defined in C: **auto**, **extern**, **register**, and **static**. The storage-class specifier of a declaration affects how the declared item is stored and initialized and which parts of a program can reference the item. Location of the declaration within the source program and the presence or absence of other declarations of the variable are also important factors in determining the visibility of variables.

Function declarations are presented in Section 4.5. For information on function definitions, see Section 7.2.

4.2 Type Specifiers

The C language provides definitions for a set of basic data types, called "fundamental" types. Their names are listed in Table 4.1.

Table 4.1

Fundamental Types

Integral Types[a]	Floating-Point Types	Other
char	**float**	**void**[c]
int	**double**	**const**
short		**volatile**[d]
long	**long double**[b]	
signed		
unsigned		
enum		

[a] The optional keyword **signed** may precede any of the integral types, except **enum**. The keyword **unsigned** may also precede any integral type except **enum**, but may also be used alone as a type specifier, in which case it is understood as **unsigned int**. When used alone, the keyword **int** is assumed to be **signed**. When used alone, the keywords **long** and **short** are understood as **long int** and **short int**.

[b] The **long double** type is semantically equivalent to **double**, but is syntactically distinct.

[c] The keyword **void** has three uses: to declare function return types, to specify that a function will take no arguments, and to modify a pointer.

[d] The **volatile** keyword is implemented syntactically, but not semantically.

Enumeration types are considered fundamental types. Type specifiers for enumeration types are discussed in Section 4.8.1.

Note

The **long float** type is no longer supported, and occurrences of it in old code should be changed to **double**.

The **signed char**, **signed int**, **signed short int**, and **signed long int** types, together with their **unsigned** counterparts, are called "integral" types. The **float**, **double**, and **long double** type specifiers refer to "floating-point" types. You can use any integral or floating-point type specifier in a variable or function declaration.

You can use the **void** type only to declare functions that return no value or to declare a pointer to an unspecified type. When the keyword **void** occurs alone within the parentheses following a function name, it is not interpreted as a type specifier. In that context **void** indicates only that the function accepts no arguments. Function types are discussed in Section 4.5.

The **const** type specifier is used to declare an object as being nonmodifiable. The **const** keyword can be used as a modifier for any fundamental or aggregate type, or to modify a pointer to an object of any type. The only type specifier that cannot be modified by **const** is **void**. A **typedef** may be modified by a **const** type specifier. A declaration that includes the keyword **const** as a modifier of an aggregate type declarator indicates that each element of the aggregate type is unmodifiable. If an item is declared with only the **const** type specifier, it's type is taken to be **const int**. A **const** object may be placed in a read-only region of storage.

The **volatile** type specifier declares an item whose value may legitimately be changed by something beyond the control of the program in which it appears. The **volatile** keyword can modify any type except **void**, including a **typedef**. An item may be both **const** and **volatile**, in which case the item could not be legitimately modified by its own program, but could be modified by some asyncronous process. The **volatile** keyword is implemented syntactically, but not semantically.

You can create additional type specifiers with **typedef** declarations, as described in Section 4.8.2. Such specifiers may only be modified by the **const** and **volatile** modifiers.

Type specifiers are commonly abbreviated, as shown in Table 4.2. Integral types are signed by default. Thus, if you omit the **unsigned** keyword from the type specifier, the integral type is signed, even if you do not specify the **signed** keyword.

In some implementations, you can specify a compiler option that changes the default **char** type from signed to unsigned. When this option is in effect, the abbreviation **char** means the same as **unsigned char**, and you must use the **signed** keyword to declare a signed character value.

Table 4.2

Type Specifiers and Abbreviations

Type Specifier	Abbreviations
signed char[a]	**char**
signed int	**signed, int**
signed short int	**short, signed short**
signed long int	**long, signed long**
unsigned char[b]	--
unsigned int	**unsigned**
unsigned short int	**unsigned short**
unsigned long int	**unsigned long**
float	--
const int	**const**
volatile int	**volatile**
const volatile int	**const volatile**

[a] When you make the **char** type unsigned by default (by specifying the appropriate compiler option), you cannot abbreviate **signed char**.

[b] When you make the **char** type unsigned by default (by specifying the appropriate compiler option), you can abbreviate **unsigned char** as **char**.

Note

This manual generally uses the abbreviated forms of the type specifiers listed in Table 4.2 rather than the long forms, and it assumes that the **char** type is signed by default. Therefore, throughout this manual, **char** stands for **signed char**.

Table 4.3 summarizes the storage associated with each fundamental type and gives the range of values that can be stored in a variable of each type. Since the **void** type specifier is only used to denote a function with no return value or a pointer to an unspecified type, it is not included in table. Similarly, the table does not include **const** because a variable type modified by **const** retains its storage size and can contain any value within range for its fundamental type.

Table 4.3

Storage and Range of Values for Fundamental Types

Type	Storage	Range of Values (Internal)
char	1 byte	– 128 to 127
int	implementation-defined	
short	2 bytes	– 32,768 to 32,767
long	4 bytes	– 2,147,483,648 to 2,147,483,647
unsigned char	1 byte	0 to 255
unsigned	implementation-defined	
unsigned short	2 bytes	0 to 65,535
unsigned long	4 bytes	0 to 4,294,967,295
float	4 bytes	IEEE-standard notation; discussed below
double	8 bytes	IEEE-standard notation; discussed below
long double	8 bytes	IEEE-standard notation; discussed below

The **char** type is used to store the integer value of a member of the representable character set. That integer value is the ASCII code corresponding to the specified character. Since the **char** type is interpreted as a signed, 1-byte integer, a **char** variable can store values in the range –128 to 127, although only the values from 0 to 127 have character equivalents. Similarly, an **unsigned char** variable can store values in the range 0 to 255.

Note that the C language does not define the storage and range associated with the **int** and **unsigned int** types. Instead, the size of a signed or unsigned **int** item is the standard size of an integer on a particular machine. For example, on a 16-bit machine the **int** type is usually 16 bits, or 2 bytes. On a 32-bit machine the **int** type is usually 32 bits, or 4 bytes. Thus, the **int** type is equivalent to either the **short int** or the **long int** type, and the **unsigned int** type is equivalent to either the **unsigned short** or the **unsigned long** type, depending on the implementation.

The type specifiers **int** and **unsigned int** (or simply **unsigned**) define certain features of the C language (for instance, the **enum** type discussed later in Section 4.8.1). In these cases, the definitions of **int** and **unsigned int** for a particular implementation determine the actual storage.

Note

The **int** and **unsigned int** type specifiers are widely used in C programs because they allow a particular machine to handle integer values in the most efficient way for that machine. However, since the sizes of the **int** and **unsigned int** types vary, programs that depend on a specific **int** size may not be portable to other machines. You can use expressions with the **sizeof** operator (discussed in Section 5.3.4) instead of hard-coded data sizes to make programs more portable. The actual sizes of **int** and **unsigned int** are discussed in your User's Guide.

Floating-point numbers use the IEEE (Institute of Electrical and Electronics Engineers, Inc.) format. Values with **float** type have 4 bytes, consisting of a sign bit, an 8-bit excess-127 binary exponent, and a 23-bit mantissa. The mantissa represents a number between 1.0 and 2.0. Since the high-order bit of the mantissa is always 1, it is not stored in the number. This representation gives a range of approximately 3.4E–38 to 3.4E+38 for type **float**.

Values with **double** type have 8 bytes. The format is similar to the **float** format except that it has an 11-bit excess-1023 exponent and a 52-bit mantissa, plus the implied high-order 1 bit. This format gives a range of approximately 1.7E–308 to 1.7E+308 for type **double**.

Range of Values

The range of values for a variable is bounded by the minimum and maximum values that can be represented *internally* in a given number of bits. However, because of C's conversion rules (discussed in detail in Chapter 5, "Expressions and Assignments"), you cannot always use the maximum or minimum value for a constant of a particular type in an expression.

For example, the constant expression `-32768` consists of the arithmetic negation operator (–) applied to the constant value 32,768. Since 32,768 is too large to represent as a **short int**, it is given the **long** type. Consequently, the constant expression `-32768` has **long** type. You can only represent –32,768 as a **short int** by type-casting it to the **short** type. No information is lost in the type cast, since –32,768 can be represented internally in 2 bytes.

Similarly, a value such as 65,000 can only be represented as an **unsigned short** by type-casting the value to **unsigned short** type or by giving the value in octal or hexadecimal notation. The value 65,000 in decimal notation is considered a signed constant, and it is given the **long** type because 65,000 does not fit into a **short**. You can cast this **long** value to the **unsigned short** type without loss of information, since 65,000 can fit in 2 bytes when it is stored as an unsigned number.

Octal and hexadecimal constants may have either **signed** or **unsigned** type, depending on their size (see Section 2.3.1, "Integer Constants," for more information). However, the method used to assign types to octal and hexadecimal constants ensures that they always behave like unsigned integers in type conversions.

Data Type Categories

The C data types fall into two general categories, called scalar and aggregate. Scalars include pointers and arithmetic types. Arithmetic types include all floating and integral types. The floating types are **float**, **double**, and **long double**. The integral types are **char**, all the variations of **int**, and the enumerated types, which include **enum** and **void**.

Aggregate types include arrays and structures.

Table 4.4 illustrates the categorization of C data types.

Table 4.4
C Data-Type Categories

Data Types	Categories		
char	Integral Types	Arithmetic Types	Scalar Types
int	Integral Types	Arithmetic Types	Scalar Types
short	Integral Types	Arithmetic Types	Scalar Types
long	Integral Types	Arithmetic Types	Scalar Types
signed	Integral Types	Arithmetic Types	Scalar Types
unsigned	Integral Types	Arithmetic Types	Scalar Types
enum	Integral Types	Arithmetic Types	Scalar Types
void	Integral Types	Arithmetic Types	Scalar Types
float	Floating Types	Arithmetic Types	Scalar Types
double	Floating Types	Arithmetic Types	Scalar Types
long double	Floating Types	Arithmetic Types	Scalar Types
Pointers			Scalar Types
Arrays			Aggregate Types
Structures			Aggregate Types

4.3 Declarators

Syntax

identifier
declarator[ ]
declarator[*constant-expression*]
**declarator*
(*declarator*)

The C language lets you declare *arrays* of values, *pointers* to values, and *functions returning* values of specified types. You must use a *declarator* to declare these items.

A declarator is an identifier that may be modified by brackets ([ ]), asterisks (*), or parentheses (( )) to declare an array, pointer, or function type, respectively. Declarators appear in the pointer, array, and function declarations described later in this chapter (Sections 4.4.6, 4.4.5, and 4.5, respectively). The following section discusses the rules for forming and interpreting declarators.

4.3.1 Pointer, Array, and Function Declarators

When a declarator consists of an unmodified identifier, the item being declared has a base type. If asterisks (*) appear to the left of an identifier, the type is modified to a *pointer* type. If the identifier is followed by brackets ([ ]), the type is modified to an *array* type. If the identifier is followed by parentheses, the type is modified to a *function returning* type.

A declarator must include a type specifier to be a complete declaration. The type specifier gives the type of the elements of an array type, the type of object addressed by a pointer type, or the return type of a function.

The sections on pointer, array, and function declarations later in this chapter discuss each type of declaration in detail (see Sections 4.4.6, 4.4.5, and 4.5, respectively).

Examples

The following examples illustrate the simplest forms of declarators:

```
/******************** Example 1 ********************/
int list[20];
```

Example 1 declares an array of **int** values named `list`.

```
/******************** Example 2 ********************/
char *cp;
```

Example 2 declares a pointer named cp to a **char** value.

```
/******************** Example 3 ********************/
double func(void);
```

Example 3 declares a function named func with no arguments that returns a **double** value.

4.3.2 Complex Declarators

You can enclose any declarator in parentheses to specify a particular interpretation of a complex declarator.

A "complex" declarator is an identifier qualified by more than one array, pointer, or function modifier. You can apply various combinations of array, pointer, and function modifiers to a single identifier. However, a declarator may not have the following illegal combinations:

- An array cannot have functions as its elements.
- A function cannot return an array or a function.

In interpreting complex declarators, brackets and parentheses (that is, modifiers to the right of the identifier) take precedence over asterisks (that is, modifiers to the left of the identifier). Brackets and parentheses have the same precedence and associate from left to right. After the declarator has been fully interpreted, the type specifier is applied as the last step. You can use parentheses to override the default association order in a way that forces a particular interpretation.

A simple way to interpret complex declarators is to read them "from the inside out," using the following four steps:

1. Start with the identifier and look to the right for brackets or parentheses (if any).
2. Interpret these brackets or parentheses, then look to the left for asterisks.
3. If you encounter a right parenthesis at any stage, go back and apply rules 1 and 2 to everything within the parentheses before proceeding.

4. Apply the type specifier.

Examples

```
/******************** Example 1 *******************/
char *(*(*var)())[10];
^    ^ ^ ^ ^   ^ ^

 7   6 4 2 1   3 5
```

In Example 1, the steps are labeled in order and can be interpreted as follows:

1. The identifier var is declared as
2. a pointer to
3. a function returning
4. a pointer to
5. an array of 10 elements, which are
6. pointers to
7. **char** values.

Examples 2 through 9 illustrate complex declarations further and show how parentheses can affect the meaning of a declaration.

```
/******************** Example 2 *******************/

     /* array of pointers to int values */
int *var[5];
```

In Example 2, the array modifier has higher priority than the pointer modifier, so var is declared to be an array. The pointer modifier applies to the type of the array elements; therefore, the array elements are pointers to **int** values.

```
/******************** Example 3 *******************/

       /* pointer to array of int values */
int (*var)[5];
```

In Example 3, parentheses give the pointer modifier higher priority than the array modifier, and `var` is declared to be a pointer to an array of five **int** values.

```
/******************** Example 4 ********************/

      /* function returning pointer to long */
long *var(long,long);
```

Function modifiers also have higher priority than pointer modifiers, so Example 4 declares `var` to be a function returning a pointer to a **long** value. The function is declared to take two **long** values as arguments.

```
/******************** Example 5 ********************/

      /* pointer to function returning long */
long (*var)(long,long);
```

Example 5 is similar to Example 3. Parentheses give the pointer modifier higher priority than the function modifier, and `var` is declared to be a pointer to a function that returns a **long** value. Again, the function takes two **long** arguments.

```
/******************** Example 6 ********************/

      /* array of pointers to functions
         returning structures */
struct both {
     int a;
     char b;
     } ( *var[5] )( struct both, struct both );
```

The elements of an array cannot be functions, but Example 6 demonstrates how to declare an array of pointers to functions instead. In this example, `var` is declared to be an array of five pointers to functions that return structures with two members. The arguments to the functions are declared to be two structures with the same structure type, `both`. Note that the parentheses surrounding `*var[5]` are required. Without them, the declaration is an illegal attempt to declare an array of functions, as shown below:

```
                /* ILLEGAL */
struct both *var[5]( struct both, struct both );
```

```
/******************** Example 7 ********************/

      /* function returning pointer
         to an array of 3 double values */
double ( *var( double (*)[3] ) )[3];
```

Example 7 shows how to declare a function returning a pointer to an array, since functions returning arrays are illegal. Here var is declared to be a function returning a pointer to an array of three **double** values. The function var takes one argument. The argument, like the return value, is a pointer to an array of three **double** values. The argument type is given by a complex abstract declarator. The parentheses around the asterisk in the argument type are required; without them, the argument type would be an array of three pointers to **double** values. For a discussion and examples of abstract declarators, see Section 4.9, "Type Names."

```
/******************** Example 8 ********************/

      /* array of arrays of pointers
         to pointers to unions */
union sign {
     int x;
     unsigned y;
     } **var[5][5];
```

A pointer can point to another pointer, and an array can contain arrays as elements, as the Example 8 shows. Here var is an array of five elements. Each element is a five-element array of pointers to pointers to unions with two members.

```
/******************** Example 9 ********************/

      /* array of pointers to arrays
         of pointers to unions */
union sign *(*var[5])[5];
```

Example 9 shows how the placement of parentheses changes the meaning of the declaration. In this example, var is a five-element array of pointers to five-element arrays of pointers to unions.

4.3.3 Declarators with Special Keywords

Your implementation of Microsoft C may include the following special keywords:

cdecl
far
fortran
huge
near
pascal

These keywords modify the meaning of variable and function declarations. See your User's Guide for a full discussion of the effects of these special keywords.

When a special keyword appears in a declarator, it modifies the item immediately to the right of the keyword. You can apply more than one special keyword to the same item. For example, you might modify a function identifier with both the **far** keyword and the **pascal** keyword. In this case, the order of the keywords does not matter (that is, **far pascal** and **pascal far** have the same effect). Thus the "binding" characteristics of the special keywords are the same as those of the type specifiers **const** and **volatile**. (Section 4.2 contains descriptions of the **const** and **volatile** keywords.)

You can also use two or more special keywords in different parts of a declaration to modify the meaning of the declaration. For example, the following declaration contains two occurrences of the **far** keyword:

```
int far * pascal far func(void);
```

In this example, the **pascal** and **far** keywords modify the function identifier `func`. The return value of `func` is declared to be a **far** pointer to an **int** value.

As in any C declaration, you can use parentheses to override the default interpretation of the declaration. The rules governing complex declarators (discussed in the Section 4.3.2) also apply to declarations that use the special keywords.

Examples

The following examples show the use of special keywords in declarations:

```
/******************** Example 1 ********************/

int huge database[65000];
```

Example 1 declares a **huge** array named database with 65,000 **int** elements. The **huge** keyword modifies the array declarator.

```
/******************** Example 2 ********************/

char * far * x;
```

In Example 2, the **far** keyword modifies the asterisk to its right, making x a **far** pointer to a pointer to **char**. This declaration is equivalent to the following declaration:

```
char * (far *x);
```

```
/******************** Example 3 ********************/

double near cdecl calc(double,double);

double cdecl near calc(double,double);
```

Example 3 shows two equivalent declarations. Both declare calc as a function with the **near** and **cdecl** attributes.

```
/******************** Example 4 ********************/

char far fortran initlist[INITSIZE];

char far *nextchar, far *prevchar, far *currentchar;
```

Example 4 also shows two declarations. The first declares a **far fortran** array of characters named initlist, and the second declares three **far** pointers named nextchar, prevchar, and currentchar. These pointers might be used to store the addresses of characters in the initlist array. Note that the **far** keyword must be repeated before each declarator.

```
/******************** Example 5 ********************/

char far *(far *getint)(int far *);
^    ^      ^      ^    ^    ^
 6   5      2      1    3    4
```

Example 5 shows a more complex declaration with several occurrences of the **far** keyword. The following procedure would be used to interpret this declaration:

1. The identifier `getint` is declared as a
2. **far** pointer to
3. a function taking
4. a single argument that is a **far** pointer to an **int** value
5. and returning a **far** pointer to a
6. **char** value

Note that the **far** keyword always modifies the item immediately to its right.

4.4 Variable Declarations

Syntax

⟦*sc-specifier*⟧ *type-specifier declarator* ⟦, *declarator*...⟧

This section describes the form and meaning of variable declarations. In particular, it explains how to declare the following:

Type of Variable	**Description**
Simple variables	Single-value variables with integral or floating-point type
Enumeration variables	Simple variables with integral type that hold one value from a set of named integer constants
Structures	Variables composed of a collection of values that may have different types

Unions	Variables composed of several values of different types, which occupy the same storage space
Arrays	Variables composed of a collection of elements with the same type
Pointers	Variables that point to other variables and contain variable locations (in the form of addresses) instead of values

In the general form of a variable declaration, *type-specifier* gives the data type of the variable and *declarator* gives the name of the variable, possibly modified to declare an array or a pointer type. The *type-specifier* can be a compound, as when the type is modified by **const**, **volatile**, or one of the special keywords described in Section 4.3.3. You can define more than one variable in a declaration by using multiple declarators, separated by commas.

The *sc-specifier* gives the storage class of the variable. In some contexts, you can initialize variables at the time you declare them. For information about storage classes and initialization, see Sections 4.6 and 4.7, respectively.

4.4.1 Simple Variable Declarations

Syntax

⟦*sc-specifier*⟧ *type-specifier identifier* ⟦**,** *identifier...*⟧**;**

The declaration of a simple variable specifies the variable's name and type. It can also specify the variable's storage class, as described in Section 4.6. The *identifier* in the declaration is the variable's name. The *type-specifier* is the name of a defined data type.

You can use a list of identifiers separated by commas (**,**) to specify several variables in the same declaration. Each identifier in the list names a variable. All variables defined in the declaration have the same type.

Examples

```
/******************** Example 1 ********************/
int x;
int const y=1;
```

Example 1 declares a simple variable named x. This variable can hold any value in the set defined by the **int** type for a particular implementation. The simple object y is declared as a constant value of type **int**. It is initialized to the value 1, and is not modifiable. If the declaration of y was not a defining declaration, it would receive an initial value of zero, and that value would be unmodifiable. The order of placement of the **int** and **const** type specifiers, relative to each other, is not significant.

```
/******************** Example 2 ********************/
unsigned long reply, flag;
```

Example 2 declares two variables named reply and flag. Both variables have **unsigned long** type and hold unsigned integral values.

```
/******************** Example 3 ********************/
double order;
```

Example 3 declares a variable named order that has **double** type, and can hold floating-point values.

4.4.2 Enumeration Declarations

Syntax

enum ⟦*tag*⟧ { *enum-list*} *identifier* ⟦, *identifier*...⟧;

enum *tag identifier* ⟦, *identifier*...⟧;

An enumeration declaration gives the name of an enumeration variable and defines a set of named integer constants (the "enumeration set"). A variable with enumeration type stores one of the values of the enumeration set defined by that type. The integer constants of the enumeration set have **int** type; thus, the storage associated with an enumeration variable is the storage required for a single **int** value.

Variables of **enum** type are treated as if they are of type **int** in all cases. They may be used in indexing expressions and as operands of all arithmetic and relational operators.

Enumeration declarations begin with the **enum** keyword and have the two forms shown at the beginning of this section.

- In the first form, the *enum-list* specifies the values and names of the enumeration set. (The *enum-list* is described in detail later in this section.) The optional *tag* is an identifier that names the enumeration type defined by the *enum-list.* The *identifier* names the enumeration variable. You can define more than one enumeration variable in a single enumeration declaration.
- The second form of enumeration declaration uses a previously defined enumeration *tag* to refer to an enumeration type defined elsewhere. The *tag* must refer to a defined enumeration type, and that enumeration type must be currently visible. Since the enumeration type is defined elsewhere, an *enum-list* does not appear in this type of declaration.

Enumeration List

An *enum-list* has the following form:

identifier ⟦ = *constant-expression*⟧
⟦, *identifier* ⟦ = *constant-expression*⟧ ... ⟧

Each *identifier* in an enumeration list names a value of the enumeration set. By default, the first *identifier* is associated with the value 0, the next *identifier* is associated with the value 1, and so on through the last *identifier* in the declaration. The name of an enumeration constant is equivalent to its value.

The optional phrase "= *constant-expression*" overrides the default sequence of values. Thus, if *identifier* = *constant-expression* appears in an *enum-list,* the *identifier* is associated with the value given by *constant-expression.* The *constant-expression* must have **int** type and can be negative. The next *identifier* in the list is associated with the value of "*constant-expression* + 1", unless you explicitly associate it with another value.

The following rules apply to the members of an enumeration set:

- An enumeration set can contain duplicate constant values. For example, you could associate the value 0 with two different identifiers named `null` and `zero` in the same set.
- The identifiers in the enumeration list must also be distinct from other identifiers with the same visibility, including ordinary variable names and identifiers in other enumeration lists.
- Enumeration tags must be distinct from other enumeration, structure, and union tags with the same visibility.

Examples

```
/******************** Example 1 ********************/

enum day {
        saturday,
        sunday = 0,
        monday,
        tuesday,
        wednesday,
        thursday,
        friday
        } workday;
```

Example 1 defines an enumeration type named `day` and declares a variable named `workday` with that enumeration type. The value 0 is associated with `saturday` by default. The identifier `sunday` is explicitly set to 0. The remaining identifiers are given the values 1 through 5 by default.

```
/******************** Example 2 ********************/

enum day today = wednesday;
```

In Example 2, a value from the set defined in Example 1 is assigned to the variable `today`. Note that the name of the enumeration constant is used to assign the value. Since the `day` enumeration type was previously declared, only the enumeration tag is necessary in this declaration.

4.4.3 Structure Declarations

Syntax

struct ⟦*tag*⟧ {*member-declaration-list*} *declarator* ⟦**,** *declarator...*⟧**;**

struct *tag declarator* ⟦**,** *declarator...*⟧**;**

A structure declaration names a structure variable and specifies a sequence of variable values (called "members" of the structure) that can have different types. A variable of that structure type holds the entire sequence defined by that type.

Structure declarations begin with the **struct** keyword and have two forms:

- In the first form, a *member-declaration-list* (described in detail below) specifies the types and names of the structure members. The optional *tag* is an identifier that names the structure type defined by the *member-declaration-list.*
- The second form uses a previously defined structure *tag* to refer to a structure type defined elsewhere. Thus, a *member-declaration-list* is not needed as long as the definition is visible. Declarations of pointers to structures and typedefs for structure types can use the structure *tag* before the structure type is defined. However, the structure definition must be encountered prior to any actual use of the typedef or pointer.

In both forms, each *declarator* specifies a structure variable. A *declarator* may also modify the type of the variable to a pointer to the structure type, an array of structures, or a function returning a structure.

Structure tags must be distinct from other structure, union, and enumeration tags with the same visibility.

Member-Declaration List

A *member-declaration-list* contains one or more variable or bit-field declarations.

Each variable declared in the *member-declaration-list* is defined as a member of the structure type. Variable declarations within the *member-declaration-list* have the same form as other variable declarations discussed in this chapter, except that the declarations cannot contain storage-class specifiers or initializers. The structure members can have any variable type: fundamental, array, pointer, union, or structure.

A member cannot be declared to have the type of the structure in which it appears. However, a member can be declared as a pointer to the structure type in which it appears as long as the structure type has a tag. This allows you to create linked lists of structures.

A bit-field declaration has the following form:

type-specifier ⟦*identifier*⟧ **:** *constant-expression***;**

The *constant-expression* specifies the number of bits in the bit field. The *type-specifier* of type **int** (**signed** or **unsigned**), and the *constant-expression* must be a non-negative integer value. Arrays of bit fields, pointers to bit fields, and functions returning bit fields are not allowed. The optional *identifier* names the bit field. Unnamed bit fields can be used as "dummy" fields, for alignment purposes. An unnamed bit field whose width is specified as 0 guarantees that storage for the member following it in the *member-declaration-list* begins on an **int** boundary.

Each *identifier* in a *member-declaration-list* must be unique within the list. However, they do not have to be distinct from ordinary variable names or from identifiers in other *member-declaration-lists.*

Note

A Microsoft extension to a allows **char** and **long** types (both **signed** and **unsigned**) for bit fields. Unnamed bit fields with base type **long** or **char** (**signed** or **unsigned**) force alignment to the the base type (**signed** or **unsigned**, **char** or **long**).

Microsoft C does not implement **signed** bit fields. The syntax is allowed, but a bitfield specified as **signed** is treated as **unsigned** in all conversions.

Storage

Structure members are stored sequentially in the order in which they are declared: the first member has the lowest memory address and the last member the highest. Storage for each member begins on a memory boundary appropriate to its type. Therefore, unnamed blanks can appear between structure members in memory.

Bit fields are not stored across boundaries of their declared type. For example, a bit field declared with **unsigned int** type is packed into the space remaining (if any), if the previous bit field was of type **unsigned int** Otherwise, it begins a new object on an **int** boundry.

Examples

```
/******************** Example 1 ********************/

struct {
           float x,y;
    } complex;
```

Example 1 defines a structure variable named `complex`. This structure has two members with **float** type, `x` and `y`. The structure type has no tag, and is therefore unnamed.

```
/******************** Example 2 ********************/

struct employee {
            char name[20];
            int id;
            long class;
    } temp;
```

Example 2 defines a structure variable named `temp`. The structure has three members: `name`, `id`, and `class`. The `name` member is a 20-element array, and `id` and `class` are simple members with **int** and **long** type, respectively. The identifier `employee` is the structure tag.

```
/******************** Example 3 ********************/

struct employee student, faculty, staff;
```

Example 3 defines three structure variables: `student`, `faculty`, and `staff`. Each structure has the same list of three members. The members are declared to have the structure type `employee`, defined in Example 2.

```
/******************** Example 4 ********************/

struct sample {
            char c;
            float *pf;
            struct sample *next;
```

```
    } x;
```

Example 4 defines a structure variable named `x`. The first two members of the structure are a **char** variable and a pointer to a **float** value. The third member, `next`, is declared as a pointer to the structure type being defined (`sample`).

```
/******************** Example 5 ********************/

struct {
            unsigned icon : 8;
            unsigned color : 4;
            unsigned underline : 1;
            unsigned blink : 1;
    } screen[25][80];
```

Example 5 defines a two-dimensional array of structures named `screen`. The array contains 2000 elements, and each element is an individual structure containing four bit-field members: `icon`, `color`, `underline`, and `blink`.

4.4.4 Union Declarations

Syntax

union ⟦*tag*⟧ {*member-declaration-list*} *declarator* ⟦, *declarator...*⟧;

union *tag declarator*⟦, *declarator...*⟧;

A union declaration names a union variable and specifies a set of variable values, called "members" of the union, that can have different types. A variable with **union** type stores one of the values defined by that type.

Union declarations have the same form as structure declarations, except that they begin with the **union** keyword instead of the **struct** keyword. The same rules govern structure and union declarations, except that bit-field members are not allowed in unions.

Storage

The storage associated with a union variable is the storage required for the largest member of the union. When a smaller member is stored, the union variable may contain unused memory space. All members are stored in the same memory space and start at the same address. The stored value is overwritten each time a value is assigned to a different member.

Examples

```
/******************** Example 1 ********************/

union sign {
        int svar;
        unsigned uvar;
    } number;
```

Example 1 defines a union with `sign` type and declares a variable named `number` that has two members: `svar`, a signed integer, and `uvar`, an unsigned integer. This declaration allows the current value of `number` to be stored as either a signed or an unsigned value. The tag associated with this union type is `sign`.

```
/******************** Example 2 ********************/

union {
        char *a, b;
        float f[20];
    } jack;
```

Example 2 defines a union variable named `jack`. The members of the union are, in order of their declaration, a pointer to a **char** value, a **char** value, and an array of **float** values. The storage allocated for `jack` is the storage required for the 20-element array `f`, since `f` is the longest member of the union. Because there is no tag associated withe the union, its type is unnamed.

```
/******************** Example 3 ********************/

union {
        struct {
                unsigned int icon : 8;
                unsigned color : 4;
        } window1;
        int screenval;
```

```
    } screen[25][80];
```

Example 3 defines a two-dimensional array of unions named `screen`. The array contains 2000 elements. Each element is an individual union with two members: `window1`,and `screenval`. The `window1` member is a structure with two bit-field members, `icon`, and `color`. The `screenval` member is an **int**. At any given time, each union element holds either the **int** represented by `screenval` or the structure represented by `window1`.

4.4.5 Array Declarations

Syntax

type-specifier declarator [*constant-expression*];
type-specifier declarator [ ];

An array declaration names the array and specifies the type of its elements. It may also define the number of elements in the array. A variable with array type is considered a pointer to the type of the array elements, as described in Section 5.2.2, "Identifiers."

Array declarations have the two forms shown at the beginning of this section. Their syntax differs as follows:

- In the first form, the *constant-expression* within the brackets defines the number of elements in the array. Each element has the type given by the *type-specifier,* which can be any type except **void**. An array element cannot be a function type.
- The second form omits the *constant-expression* in brackets. You can use this form only if you have initialized the array, declared it as a formal parameter, or declared it as a reference to an array explicitly defined elsewhere in the program.

In both forms, the *declarator* names the variable and may modify the variable's type. The brackets ([ ]) following the *declarator* modify the declarator to array type.

You can define an array of arrays (a "multidimensional" array), by following the array declarator with a list of bracketed *constant-expressions,* as shown below:

type-specifier declarator[*constant-expression*] [*constant-expression*] ...

Each *constant-expression* in brackets defines the number of elements in a given dimension: two-dimensional arrays have two bracketed expressions, three-dimensional arrays have three, and so on. When you declare a multidimensional array within a function, you can omit the first *constant-expression* if you have initialized the array, declared it as a formal parameter, or declared it as a reference to an array explicitly defined elsewhere in the program.

You can define arrays of pointers to various types of objects by using complex declarators, as described in Section 4.3.2.

Storage

The storage associated with an array type is the storage required for all of its elements. The elements of an array are stored in contiguous and increasing memory locations, from the first element to the last. No blanks separate the array elements in storage.

Arrays are stored by row. For example, the following array consists of two rows with three columns each:

```
char A[2][3];
```

The three columns of the first row are stored first, followed by the three columns of the second row. This means that the last subscript varies most quickly.

To refer to an individual element of an array, use a subscript expression, discussed in Section 5.2.5.

Examples

```
/******************** Example 1 ********************/

int scores[10], game;
```

Example 1 defines an array variable named `scores` with 10 elements, each of which has **int** type. The variable named `game` is declared as a simple variable with **int** type.

```
/******************** Example 2 ********************/

float matrix[10][15];
```

Example 2 defines a two-dimensional array named `matrix`. The array has 150 elements, each having **float** type.

```
/******************** Example 3 ********************/

struct {
            float x,y;
            } complex[100];
```

Example 3 defines an array of structures. This array has 100 elements; each element is a structure containing two members.

```
/******************** Example 4 ********************/

extern char *name[];
```

Example 4 declares the type and name of an array of pointers to **char**. The actual definition of name occurs elsewhere.

4.4.6 Pointer Declarations

Syntax

type-specifier * ⟦*modification-spec*⟧ *declarator*;

A pointer declaration names a pointer variable and specifies the type of the object to which the variable points. A variable declared as a pointer holds a memory address.

The *type-specifier* gives the type of the object, which can be any fundamental, structure, or union type. Pointer variables can also point to functions, arrays, and other pointers. (For information on declaring more complex pointer types, refer to Section 4.3.2, "Complex Declarators.")

The *type-specifier* can also be **void**, so that specification of the type to which the pointer points can be delayed. This is referred to as a "pointer to **void**" (**void** *), and is used to delay specification of the type to which the pointer will refer. A variable declared as a pointer to **void** can be used to point to an object of any type. However, in order to perform operations on the pointer or on the object to which it points, the type to which it points must be explicitly specified for each operation. Such conversion can be accomplished with a type cast.

The *modification-spec* can be either **const** or **volatile**, or both. These specify, respectively, that the pointer will not be modified by the program itself (**const**), or that the pointer may legitimately be modified by some process beyond the control of the program (**volatile**). (See Section 4.2 for more information on **const** and **volatile**.

The *declarator* names the variable and can include a type modifier. For example, if the *declarator* represents an array, the type of the pointer is modified to pointer to array.

You can declare a pointer to a structure or union type before you define the structure or union type. However the structure must be defined before the pointer can be dereferenced. You declare the pointer by using the structure or union tag (see Example 7 below). Such declarations are allowed because the compiler does not need to know the size of the structure or union to allocate space for the pointer variable.

Storage

The amount of storage required for an address and the meaning of the address depend on the implementation of the compiler. Pointers to different types are not guaranteed to have the same length.

In some implementations, you can use the special keywords **near**, **far**, and **huge** to modify the size of a pointer. Declarations using special keywords are described in Section 4.3.3. See your User's Guide for more information on the meaning and use of these keywords.

Examples

```
/******************** Example 1 ********************/
char *message;/
```

Example 1 defines a pointer variable named `message`. It points to a variable with **char** type.

```
/******************** Example 2 ********************/
int *pointers[10]
```

Example 2 defines an array of pointers named `pointers`. The array has 10 elements; each element is a pointer to a variable with **int** type.

```
/******************** Example 3 ********************/
int (*pointer)[10];
```

Example 3 defines a pointer variable named `pointer`; it points to an array with 10 elements. Each element in this array has **int** type.

```
/******************* Example 4 *********************/
const int *x;
```

Example 4 declares a pointer variable x, to a constant value. The pointer may be modified to point to a different integer, but the value to which it points may not be modified.

```
/****************** Example 5 *********************/
const int some_object = 5 ;
int other_object = 37;
int *const y = &fixed_object;
const volatile *const z = &some_object;
const *const volatile w = &some_object;
```

The variable y in Example 5 is declared as a constant pointer to an integer value. The value it points to may be modified, but the pointer itself must always point to the same location, the address of `fixed_object.` Similarly z is a constant pointer, but it is also declared to point to an **int** whose value will not be modified by the program. The additional specifier `volatile` indicates that although the value of the **const int** pointed to by z cannot be modified by the program, it could legitimately be modified by a process outside the program. The declaration of w specifies that the value pointed to will not be changed, and that the program itself will not modifiy the pointer. However, some outside process could legitimately modify the pointer.

```
/******************* Example 6 ********************/
struct list *next, *previous;
```

Example 6 defines two pointer variables that point to the structure type `list`. This declaration can appear before the definition of the `list` structure type (see Example 7), as long as the `list` type definition has the same visibility as the declaration.

```
/******************** Example 7 ********************/
struct list {
            char *token;
            int count;
            struct list *next;
    } line;
```

Example 7 declares the variable `line` to have the structure type named `list`. The `list` structure type is defined to have three members: the first member is a pointer to a **char** value, the second is an **int** value, and the third is a pointer to another `list` structure.

```
/******************** Example 8 ********************/
struct id {
     unsigned int id_no;
     struct name *pname;
    } record;
```

Example 8 declares the variable `record` to have the structure type `id`. Note that `pname` is declared as a pointer to another structure type named `name`. This declaration can appear before the `name` type is defined.

```
/*********************** Example 9 ********************/
int i;
void *p;            /* p declared as pointer to an object
                       whose type is not specified       */
p = &i;             /* address of integer i assigned to p
                       but type of p itself is still not
                       specified. An operation like p++
                       would not be permitted yet        */
(int *)p++;         /* incrementing p permitted when the
                       cast converts it to pointer to int */
```

The pointer variable `p` is declared in Example 9, but the **void *** preceding the identifier `p` in the declaration, means that `p` can be used later to point to any type object. The address of an **int** is assigned to it, but no operations on the pointer itself are permitted unless it is explicitly converted to the type to which it points. Similarly, indirect operations on the object dereferenced by `p` are not permitted unless `p` is converted to a specific type. Finally, `p` is converted to a pointer to **int** with a cast, and incremented.

4.5 Function Declarations (Prototypes)

Syntax

⟦*sc-spec*⟧ ⟦*type-spec*⟧ *declarator*(⟦*formal-parameter-list*⟧) ⟦, *declarator-list* ...⟧;

A function declaration, also called a "function prototype," establishes the name and return type of a function and may specify the types, formal parameter names, and number of arguments to the function. A function declaration does not define the function body. It simply makes those attributes of the function it does include known to the compiler. This information enables the compiler to check the types of the actual arguments in ensuing calls to the function.

If you do not provide a function prototype, the compiler constructs one from the first reference to the function it encounters, whether a call or a function definition. This prototype is then used to check the formal parameters in a subsequent definition of the function or the actual arguments in a subsequent call to the function. However, such checking can only be done if the definition occurs in the same source file. If the definition occurs in a different module, argument mismatch errors are not detected. Function definitions are described in detail in Section 7.2.

The *sc-spec* represents a storage-class specifier, and can be either **extern** or **static**. Storage-class specifiers are discussed in Section 4.6.

The *type-spec* gives the function's return type, and the *declarator* names the function. If you omit the *type-spec* from a function declaration, the function is assumed to return a value of type **int**.

The *formal-parameter-list* is described below.

The final *declarator-list* indicated in the syntax represents further declarations on the same line. These may be other functions returning values of the same type as the first function, or declarations of any variables whose type is the same as the first function's return type. Each such declaration must be separated from its predecessors and successors by a comma.

Formal Parameters

Formal parameters describe the actual arguments that can be passed to a function. In the function declaration, the parameter declarations establish the number and types of the actual arguments. They may also include identifiers of the formal parameters. Though the parameters may be omitted from the function declaration, their inclusion is recommended. The extent of the information in the declaration influences the argument checking done on function calls appearing before the compiler has processed the function definition.

Note that identifiers used to name the parameters in the prototype declaration are descriptive only. They go out of scope at the end of the declaration. Therefore, they need not be identical to the identifiers used in the declaration portion of the function definition. Using the same names may enhance readability, but has no other significance.

Return Type

Functions can return values of any type except arrays and functions. Therefore, the *type-specifier* of a function declaration can specify any fundamental, structure, or union type. You can modify the function identifier with one or more asterisks (*) to declare a pointer return type.

Although functions cannot return arrays and functions, they can return pointers to arrays and functions. You declare a function that returns a pointer to an array or function type by modifying the function identifier with asterisks (*), brackets ([ ]), and parentheses (( )). Such a function identifier is known as a a "complex declarator." Rules for forming and interpreting complex declarators are discussed in Section 4.3.2.

The List of Formal Parameters

All elements of the *formal-parameter-list* appearing within the parentheses following the function declarator are optional, as shown in the following syntax:

⟦⟦**void**⟧¦⟦**register**⟧ ⟦*type-spec*⟧ ⟦*declarator*⟦⟦, ...⟧⟦,...⟧⟧⟧⟧

If formal parameters are omitted from the function declaration, the parentheses can contain the keyword **void** to specify that no arguments will ever be passed to the function. If the parentheses are left entirely empty, no information is coveyed about whether arguments will be passed to the function and no checking of argument types is performed.

Note

Empty parentheses in a function declaration or definition represent an obsolescent form not recommended for new code. Functions accepting no arguments should be declared with the **void** keyword replacing the list of formal parameters. This use of **void** is interpreted by context, and should not be confused with uses of void as a type specifier.

A declaration in the list of formal parameters can contain the **register** storage-class specifier, either alone or combined with a type specifier and an identifier. If **register** is not specified, the storage class is **auto**. The only explicit storage class specifier permitted is **register**. If the parentheses contain only the **register** keyword, the formal parameter is considered to represent an unnamed **int** for which **register** storage is being requested.

If *type-spec* is included, it can specify the type name for any fundamental, structure, or union type (such as **int** for integer type). A *declarator* for a fundamental, structure, or union type is simply an identifier of a variable having that type.

The *declarator* for a pointer, array, or function can be formed by combining a type specifier, plus the appropriate modifier, with an identifier. Alternatively, an "abstract declarator" (that is, a declarator without a specified identifier) can be used. Section 4.9, "Type Names," explains how to form and interpret abstract declarators.

A full, partial, or empty list of formal parameters can be declared. If the list contains at least one declarator, a variable number of parameters can be specified by ending the list with a comma followed by three periods (**,...**),**referred**to**as**the"**ellipsis** A function is expected to have at least as many arguments as there are declarators or type specifiers preceding the last comma.

Note

To maintain compatibility with previous versions, the compiler accepts a comma without trailing periods at the end of a declarator list to indicate a variable number of arguments. However, this is a Microsoft extension to the ANSI C standard. New code should use the comma followed by three periods. For information on enabling and disabling extensions, see your User's Guide.

One other special construction is permitted as a formal parameter: **void *** represents a pointer to an object of unspecified type. Thus, in a call, the pointer can be used to reference any type of object by converting the pointer (for example, with a cast) to a pointer to the desired type. Note that before operations can be performed on the pointer or its object, the pointer must be explicitly converted. Section 4.4.6 provides further information on **void ***.

Summary

Function prototypes are optional. If included, the only elements absolutely required are the name of the function, the opening and closing parentheses following the name, and the final semicolon. If no return type is included, as in the following example, the function is assumed to return an **int**.

```
/***** Obsolescent form of function definition *****/
minimal_declaration();      /* may or may not
                               accept arguments */
```

Any appropriate combination of elements is permitted among the parameters declarations, from no information (as in the obsolescent form in the example above) to a full prototype of the function. If no prototype at all is given, a /fIde facto prototype is constructed from information in the first reference to the function encountered in the source file.

Example

```
double func(void);          /* returns a double, but
                             * accepts no arguments
                            */
fun (void *);               /* passes an unnamed pointer
                             * to an unspecified
                             * type; returns an int
                            */
char *fu(long, long);       /* passes two unnamed longs;
                             * returns pointer to char
                            */
foo(register a, char *);    /* passes a named int with request
```

```
                                  * for register storage, and an
                                  * unnamed pointer to char;
                                  * returns an int
                                 */
void go(int *[], char *b);       /* passes pointer to an unnamed
                                  * array of int using an abstract
                                  * declarator, and a pointer to char
                                  * named b; there is no return
                                 */
void *tu(double v,...);          /* passes at least one double named
                                  * v; other parameters may also be
                                  * passed; returns a pointer
                                  * to an unspecified type
                                 */
```

The compiler uses any information included in the parameter list to check any actual arguments appearing before the compiler has processed the function definition.

Examples

```
/******************** Example 1 ********************/
int add(int num1, int num2);
```

Example 1 declares a function named `add` that takes two **int** arguments, represented by the identifiers `num1` and `num2`, and returns an **int** value.

```
/******************** Example 2 ********************/
double calc();
```

Example 2 declares a function named `calc` that returns a **double** value. The obsolescent empty parentheses leave the issue of possible arguments to the function undefined.

```
/******************** Example 3 ********************/
char *strfind(char *ptr,...);
```

Example 3 declares a function named `strfind`, that returns a pointer to **char**. The function accepts at least one argument, declared by the formal parameter `char *ptr`, to be a pointer to a **char** value. The list of argument types has one entry, and ends with a comma followed by three periods, indicating that the function may take more arguments.

```
/******************** Example 4 ********************/
```

```
void draw(void);
```

Example 4 declares a function with **void** return type (returning no value). The *void* keyword also replaces the list of formal parameters, so no arguments are expected for this function.

```
/******************** Example 5 ********************/
double (*sum(double, double))[3];
```

In Example 5, sum is declared as a function returning a pointer to an array of three **double** values. The sum function takes two unnamed **double** values as arguments.

```
/******************** Example 6 ********************/
int (*select(void))(int number);
```

In Example 6, the function named `select` is declared to take no arguments and to return a pointer to a function. The pointer return value points to a function taking one **int** argument, represented by the identifier `number`, and returning an **int** value.

```
/******************** Example 7 ********************/
int prt(void *);
```

In Example 7, the function `prt` is declared to take a pointer argument of any type and return an **int**. A pointer to any type could be passed as an argument to `prt` without producing a type-mismatch warning.

```
/******************** Example 8 ***********************/
long (*const rainbow[]) (int, ...) ;
```

Example 8 shows the declaration of an array named `rainbow`, of an unspecified number of constant pointers to functions, each of which passes at least one parameter of type **int**, as well as an unspecified number of other parameters. Each of the functions pointed to returns a **long** value.

4.6 Storage Classes

The storage class of a variable determines whether the item has a "global" or "local" lifetime. An item with a global lifetime exists and has a value throughout the duration of the program. All functions have global lifetimes.

Variables with local lifetimes are allocated new storage each time execution control passes to the block in which they are defined. When execution control passes out of the block, the variables no longer have meaningful values.

Although C defines only two types of storage classes, it provides the following four storage-class specifiers:

auto
register
static
extern

Items declared with the **auto** or **register** specifier have local lifetimes. Items declared with the **static** or **extern** specifier have global lifetimes.

The four storage-class specifiers have distinct meanings because storage-class specifiers affect the visibility of functions and variables, as well as their storage class. The term "visibility" refers to the portion of the source program in which the variable or function can be referenced by name. An item with a global lifetime exists throughout the execution of the source program, but it may not be "visible" in all parts of the program. (Visibility and the related concept of lifetime are discussed in Chapter 3, "Program Structure.")

The placement of variable and function declarations within source files also affects storage class and visibility. Declarations outside all function definitions are said to appear at the "external level"; declarations within function definitions appear at the "internal level."

The exact meaning of each storage-class specifier depends on two factors:

- Whether the declaration appears at the external or internal level
- Whether the item being declared is a variable or a function

Sections 4.6.1 – 4.6.3 describe the meanings of storage-class specifiers in each kind of declaration and explain the default behavior when the storage-class specifier is omitted from a variable or function declaration.

4.6.1 Variable Declarations at the External Level

In variable declarations at the external level (that is, outside all functions), you can use the **static** or **extern** storage-class specifier or omit the storage-class specifier entirely. You cannot use the **auto** and **register** storage-class specifiers at the external level.

Variable declarations at the external level are either *definitions* of variables ("defining declarations"), or *references* to variables defined elsewhere ("referencing declarations").

An external variable declaration that also initializes the variable (implicitly or explicitly) is a defining definition of the variable. Definitions at the external level can take several forms:

- A variable that you declare with the **static** storage-class specifier. You can explicitly initialize the **static** variable with a constant expression, as described in Section 4.7. If you omit the initializer, the variable is initialized to 0 by default. For example, `static int k = 16;` and `static int k;` are both considered definitions of the variable `k`.
- A variable that you explicitly initialize at the external level. For example, `int j = 3;` is a definition of the variable `j`.

Once a variable is defined at the external level, it is visible throughout the rest of the source file in which it appears. The variable is not visible prior to its definition in the same source file. Also, it is not visible in other source files of the program, unless a referencing declaration makes it visible, as described below.

You can define a variable at the external level only once within a source file. If you give the **static** storage-class specifier, you can define another variable with the same name and the **static** storage-class specifier in a different source file. Since each **static** definition is visible only within its own source file, no conflict occurs.

The **extern** storage-class specifier declares a *reference* to a variable defined elsewhere. You can use an **extern** declaration to make a definition in another source file visible, or to make a variable visible above its definition in the same source file. Once you have declared a reference to the variable at the external level, the variable is visible throughout the remainder of the source file in which the declared reference occurs.

Declarations that use the **extern** storage-class specifier cannot contain initializers, since these declarations refer to variables whose values are defined elsewhere.

For an **extern** reference to be valid, the variable it refers to must be defined once, and only once, at the external level. The definition can be in any of the source files that form the program.

One special case is not covered by the rules outlined above. You can omit both the storage-class specifier and the initializer from a variable declaration at the external level; for example, the declaration `int n;` is a valid external declaration. This declaration can have one of two different meanings, depending on the context:

1. If there is an external defining declaration of a variable with the same name elsewhere in the program, the current declaration is assumed to be a reference to the variable in the defining declaration, exactly as if the **extern** storage-class specifier had been used in the declaration.
2. If there is no external defining declaration of a variable with the same name elsewhere in the program, the declared variable is allocated storage at link time and initialized to 0. This kind of variable is known as a "communal" variable. If more than one such declaration appears in the program, storage is allocated for the largest size declared for the variable. For example, if a program contains two uninitialized declarations of `i` at the external level, `int i;` and `char i;`, storage space for an **int** is allocated for `i` at link time.

Uninitialized variable declarations at the external level are not recommended for any file that might be placed in a library.

Example

```
/*************************************************************
                    SOURCE FILE ONE
*************************************************************/

extern int i;                     /* reference to i,
                                     defined below */

main()
{
        i++;
        printf("%d\n", i);    /* i equals 4 */
        next();
}

int i = 3;                    /* definition of i */

next()
{
        i++;
        printf("%d\n", i);    /* i equals 5 */
        other();
}

/**************************************************************
                SOURCE FILE TWO
**************************************************************/

extern int i;                 /* reference to i in
                                 first source file */

other()
{
        i++;
        printf("%d\n", i);    /* i equals 6 */
}
```

The two source files in this example contain a total of three external declarations of `i`. Only one declaration contains an initialization; that declaration, `int i = 3;` , defines the global variable `i` with initial value 3. The **extern** declaration of `i` at the top of the second source file makes the global variable visible above its definition in the file. Without the **extern** declaration, the `main` function could not reference the global variable `i`. The **extern** declaration of `i` in the second source file also makes the global variable visible in that source file.

Assuming that the `printf` function is defined elsewhere in the program, all three functions perform the same task: they increase `i` and print it. The values 4, 5, and 6 are printed.

If the variable `i` had not been initialized, it would have been set to 0 automatically at link time. In this case, the values 1, 2, and 3 would have been printed.

4.6.2 Variable Declarations at the Internal Level

You can use any of the four storage-class specifiers for variable declarations at the internal level. When you omit the storage-class specifier from such a declaration, the default storage class is **auto**.

The **auto** storage-class specifier declares a variable with a local lifetime. An **auto** variable is visible only in the block in which it is declared. Declarations of **auto** variables can include initializers, as discussed in Section 4.7. Since variables with **auto** storage class are not initialized automatically, you should either explicitly initialize them when you declare them or assign them initial values in statements within the block. The values of uninitialized **auto** variables are undefined.

A **static auto** variable can be initialized with the address of any external or **static** item, but not with the address of another **auto** item, because the address of an **auto** item is not a constant.

The **register** storage-class specifier tells the compiler to give the variable storage in a register, if possible. Register storage usually speeds access time and reduces code size. Variables declared with **register** storage class have the same visibility as **auto** variables. The number of registers that can be used for variable storage is machine-dependent. If no registers are available when the compiler encounters a **register** declaration, the variable is given **auto** storage class and stored in memory. The compiler assigns register storage to variables in the order in which the declarations appear in the source file. Register storage, if available, is only guaranteed for **int** and pointer types that are the same size as an **int**.

A variable declared at the internal level with the **static** storage-class specifier has a global lifetime but is visible only within the block in which it is declared. Unlike **auto** variables, **static** variables keep their values when the block is exited. You can initialize a **static** variable with a constant expression. A **static** variable is initialized only once, when program execution begins; it is *not* reinitialized each time the block is entered. If you do not explicitly initialize a **static** variable, it is initialized to 0 by default.

A variable declared with the **extern** storage-class specifier is a reference to a variable with the same name defined at the external level in any of the source files of the program. The internal **extern** declaration is used to make the external-level variable definition visible within the block. Unless otherwise declared at the external level, a variable declared with the **extern** keyword is visible only in the block in which it is declared.

Example

```
int i = 1;

main()
{       /* reference to i, defined above: */
        extern int i;

        /* initial value is zero; a is
           visible only within main: */
        static int a;

        /* b is stored in a register, if possible: */
        register int b = 0;

        /* default storage class is auto: */
        int c = 0;

        /* values printed are 1, 0, 0, 0: */
        printf("%d\n%d\n%d\n%d\n", i, a, b, c);
        other();
}

other()
{
        /* address of global i assigned to pointer variable */
        static int *external_i = &i;

        /* i is redefined; global i no longer visible: */
        int i = 16;

        /* this a is visible only within other: */
        static int a = 2;

        a += 2;
        /* values printed are 16, 4, and 1: */
        printf("%d\n%d\n%d\n", i, a, *external_i);
}
```

In this example, the variable `i` is defined at the external level with initial value 1. An **extern** declaration in the `main` function is used to declare a reference to the external-level `i`. The **static** variable a is initialized to 0 by default, since the initializer is omitted. The call to `printf` (assuming the `printf` function is defined elsewhere in the source program) prints the values 1, 0, 0, and 0.

In the `other` function, the address of the global variable `i` is used to initialize the **static** pointer variable `external_i`. This works because the global variable has **static** lifetime, meaning its address will always be the same. Next, the variable `i` is redefined as a local variable with initial value

16. This redefinition does not affect the value of the external-level i, which is hidden by the use of its name for the local variable. The value of the global i is now accessible only indirectly within this block, through the pointer external_i. Attempting to assign the address of the **auto** variable i to a pointer would not work, since it may be different each time the block is entered. The variable a is declared as a **static** variable and initialized to 2. This a does not conflict with the a in main, since **static** variables at the internal level are visible only within the block in which they are declared.

The variable a is increased by 2, giving 4 as the result. If the other function were called again in the same program, the initial value of a would be 4, since internal **static** variables keep their values when the program exits and then re-enters the block in which they are declared.

4.6.3 Function Declarations at the External and Internal Levels

You can use either the **static** or the **extern** storage-class specifier in function declarations. Functions always have global lifetimes.

The visibility rules for functions vary slightly from the rules for variables, as follows:

- A function declared to be **static** is visible only within the source file in which it is defined. Functions in the same source file can call the **static** function, but functions in other source files cannot. You can declare another **static** function with the same name in a different source file without conflict.
- Functions declared as **extern** are visible throughout all the source files that make up the program (unless you later redeclare such a function as **static**). Any function can call an **extern** function.
- Function declarations that omit the storage-class specifier default to **extern**.

4.7 Initialization

Note

A Microsoft extension to the ANSI C standard provides that function declarations at the internal level have the same meaning as function declarations at the external level. This means that a function is visible from its point of declaration through the rest of the source file.

Syntax

= *initializer*

You can set a variable to an initial value by applying an initializer to the declarator in the variable declaration. The value or values of the initializer are assigned to the variable. The initializer is preceded by an equal sign.

You can initialize variables of any type, provided that you obey the following rules:

- You cannot use initializers in declarations that use the **extern** storage-class specifier.
- You can initialize variables declared at the external level. If you do not explicitly initialize a variable at the external level, it is initialized to 0 by default.
- You can use a constant expression to initialize any variable declared with the **static** storage-class specifier. Variables declared to be **static** are initialized, when program execution begins. If you do not explicitly initialize a **static** variable, it is initialized to 0 by default.
- Variables declared with the **auto** and **register** storage-class specifiers are initialized each time execution control passes to the block in which they are declared. If you omit an initializer from the declaration of an **auto** or **register** variable, the initial value of the variable is undefined.
- You cannot initialize **auto** aggregate types (arrays, structures, and unions). Only **static** aggregates and aggregates declared at the external level can be initialized.

- The initial values for external variable declarations and for all **static** variables, whether external or internal, must be constant expressions. (Constant expressions are described in Section 5.2.10.) You can use either constant or variable values to initialize **auto** and **register** variables.

Sections 4.7.1 and 4.7.2 describe how to initialize variables of fundamental, pointer, and aggregate types.

4.7.1 Fundamental and Pointer Types

Syntax

= *expression*

The value of *expression* is assigned to the variable. The conversion rules for assignment apply.

An internally-declared static variable can only be initialized with a constant value. Since the address of any externally declared or static variable is constant, it may be used to initialize an internally-declared **static** pointer variable. However, the address of an **auto** variable cannot be used as an initializer because it may be different for each execution of the block.

Examples

```
/******************** Example 1 ********************/
int x = 10;
```

In Example 1, x is initialized to the constant expression 10.

```
/******************** Example 2 ********************/
register int *px = 0;
```

In Example 2, the pointer px is initialized to 0, producing a "null" pointer.

```
/******************** Example 3 ********************/
const int c = (3 * 1024);
```

Example 3 uses a constant expression to initialize c to a constant value that cannot be modified.

```
/******************** Example 4 ********************/
int *b = &x;
int *const a = &z;
```

Example 4 initializes the pointer b with the address of another variable, x. The pointer a is initialized with the address of a variable named z; However, since it is specified to be a **const** pointer, the variable a can only be initialized, never modified. It always points to the same location.i

```
/******************** Example 5 ********************/

int GLOBAL ;

int function(void)
  {
     int LOCAL ;
     static int *lp = &LOCAL  /* Illegal declaration */
     static int *gp = &GLOBAL /* Legal declaration */
  }
```

The global variable GLOBAL is declared in Example 5 at the external level, so it has global lifetime. The local variable LOCAL has **auto** storage class and only has an address during the execution of the function in which is is declared. Therefore, attempting to initialize the **static** pointer variable lp with the address of LOCAL is not permitted. The **static** pointer variable gp can be initialized to the address of GLOBAL because that address is always the same.

4.7.2 Aggregate Types

Syntax

= { *initializer-list*}

An *initializer-list* is a list of initializers separated by commas. Each initializer in the list is either a constant expression or an *initializer-list*. Therefore, an *initializer-list* enclosed in braces can appear within another *initializer-list*. This form is useful for initializing aggregate members of an aggregate types as shown in the examples below.

For each *initializer-list*, the values of the constant expressions are assigned, in order, to the corresponding members of the aggregate variable. When a union is initialized, the *initializer-list* must be a single constant expression. The value of the constant expression is assigned to the first member of the union.

If an *initializer-list* has fewer values than an aggregate type, the remaining members or elements of the aggregate type are initialized to 0. If an *initializer-list* has more values than an aggregate type, an error results. These rules apply to each embedded *initializer-list*, as well as to the aggregate as a whole.

For example,

```
int P[4][3] = {
        { 1, 1, 1 },
        { 2, 2, 2 },
        { 3, 3, 3,},
        { 4, 4, 4,},
};
```

declares `P` as a 4-by-3 array and initializes the elements of its first row to 1, the elements of its second row to 2, and so on through the fourth row. Note that the *initializer-list* for the third and fourth rows contains commas after the last constant expression. The last *initializer-list* (`{4, 4, 4,}`) is also followed by a comma. These extra commas are permitted but are not required; only commas that separate constant expressions from one another, and those that separate one *initializer-list* from another, are required.

If there is no embedded initializer list for an aggregate member, values are simply assigned, in order, to each member of the subaggregate. Therefore, the initialization in the previous example is equivalent to the following:

```
int P[4][3] = {
        1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4
};
```

Braces can also appear around individual initializers in the list.

When you initialize an aggregate variable, you must be careful to use braces and initializer lists properly. The following example illustrates the compiler's interpretation of braces in more detail:

```
typedef struct {
    int n1, n2, n3;
} triplet;

triplet nlist[2][3] = {
    { {  1, 2, 3 }, {  4, 5, 6 }, {  7, 8, 9 } },   /* Line 1 */
    { { 10,11,12 }, { 13,14,15 }, { 15,16,17 } }    /* Line 2 */
};
```

In this example, `nlist` is declared as a 2-by-3 array of structures, each structure having three members. Line 1 of the initialization assigns values to the first row of `nlist`, as follows:

1. The first left brace on Line 1 signals the compiler that initialization of the first aggregate member of `nlist` (that is, `nlist[0]`) is beginning.
2. The second left brace indicates that initialization of the first aggregate member of `nlist[0]` (that is, the structure at `nlist[0][0]`) is beginning.
3. The first right brace ends initialization of the structure `nlist[0][0]`; the next left brace starts initialization of `nlist[0][1]`.
4. The process continues until the end of the line, where the closing right brace ends initialization of `nlist[0]`.

Line 2 assigns values to the second row of `nlist` in a similar way.

Note that the outer sets of braces enclosing the initializers on lines 1 and 2 are required. The following construction, which omits the outer braces, would cause an error:

```
     /* THIS CAUSES AN ERROR */

triplet nlist[2][3] = {
     {  1, 2, 3 },{  4, 5, 6 },{  7, 8, 9 },    /* Line 1 */
     { 10,11,12 },{ 13,14,15 },{ 16,17,18 }     /* Line 2 */
};
```

In this construction, the first left brace on line 1 starts the initialization of `nlist[0]`, which is an array of three structures. The values 1, 2, and 3 are assigned to the three members of the first structure. When the next right brace is encountered (after the value 3), initialization of `nlist[0]` is complete, and the two remaining structures in the three-structure array are automatically initialized to 0. Similarly, `{ 4,5,6 }` initializes the first structure in the second row of `nlist`. The remaining two structures of `nlist[1]` are set to 0. When the compiler encounters the next initializer list (`{ 7,8,9 }`), it tries to initialize `nlist[2]`. Since `nlist` has only two rows, this attempt causes an error.

Examples

```
/******************** Example 1 ********************/

struct list {
        int i, j, k;
        float m[2][3];
        } x = {
                1,
                2,
                3,
                {4.0, 4.0, 4.0}
        };
```

In Example 1, the three **int** members of x are initialized to 1, 2, and 3, respectively. The three elements in the first row of m are initialized to 4.0; the elements of the remaining row of m are initialized to 0.0 by default.

```
/******************** Example 2 ********************/

union
 {
        char x[2][3];
        int i, j, k;
        } y = { {
                 {'1'},
                 {'4'} }
        };
```

In Example 2, the union variable y is initialized. The first element of the union is an array, so the initializer is an aggregate initializer. The initializer list {'1'} assigns values to the first row of the array. Since only one value appears in the list, the element in the first column is initialized to the character 1, and the remaining two elements in the row are initialized to zero by default. Similarly, the first element of the second row of x is initialized to the character 4, and the remaining two elements in the row are initialized to zero.

4.7.3 String Initializers

Syntax

= "*characters*"

You can initialize an array of characters with a string literal. For example,

```
char code[ ] = "abc";
```

initializes code as a four-element array of characters. The fourth element is the null character that terminates all string literals.

If you specify the array size and the string is longer than the specified array size, the extra characters are simply ignored. For example, the following declaration initializes code as a three-element character array:

```
char code[3] = "abcd";
```

Only the first three characters of the initializer are assigned to code. The character d and the string-terminating null character are discarded. Beware that this creates an unterminated string(that is, one without a zero value to mark its end), and generates a diagnostic message indicating the condition.

If the string is shorter than the specified array size, the remaining elements of the array are initialized to zero values.

4.8 Type Declarations

A type declaration defines the name and members of a structure or union type, or the name and enumeration set of an enumeration type. You can use the name of a declared type in variable or function declarations to refer to that type. This is useful if many variables and functions have the same type.

A **typedef** declaration defines a type specifier for a type. You can use **typedef** declarations to construct shorter or more meaningful names for types already defined by C or for types that you have declared.

4.8.1 Structure, Union, and Enumeration Types

Declarations of structure, union, and enumeration types have the same general form as variable declarations of those types. However, type declarations and variable declarations differ in the following ways:

- In type declarations the variable identifier is omitted, since no variable is declared.
- In type declarations the *tag* is required; it names the structure, union, or enumeration type.
- The *member-declaration-list* or *enum-list* defining the type must appear in the type declaration; the abbreviated form of variable declarations, in which a *tag* refers to a type defined elsewhere, is not legal for type declarations.

Examples

```
/******************** Example 1 ********************/

enum status {
        loss = -1,
        bye,
        tie = 0,
        win
        };
```

Example 1 declares an enumeration type named `status`. The name of the type can be used in declarations of enumeration variables. The identifier `loss` is explicitly set to −1. Both `bye` and `tie` are associated with the value 0, and `win` is given the value 1.

```
/******************** Example 2 ********************/

struct student {
        char name[20];
        int id, class;
        };
```

Example 2 declares a structure type named `student`. A declaration such as `struct student employee;` can be used to declare a structure variable with `student` type.

4.8.2 Typedef Declarations

Syntax

typedef *type-specifier declarator* ⟦**,** *declarator...*⟧**;**

A **typedef** declaration is analogous to a variable declaration except that the **typedef** keyword replaces a storage-class specifier. A **typedef** declaration is interpreted in the same way as a variable or function declaration, but the identifier, instead of assuming the type specified by the declaration, becomes a synonym for the type.

Note that a **typedef** declaration does not create types. It creates synonyms for existing types, or names for types that could be specified in other ways. When a **typedef** name is used as a type specifier, it can be combined with certain type specifiers, but not others. Acceptable modifiers include **const**,**volatile**. In some implementations there are additional special keywords that can be used to modify a **typedef**. (The special keywords are described in Section 4.3.3.)

You can declare any type with **typedef**, including pointer, function, and array types. You can declare a **typedef** name for a pointer to a structure or union type before you define the structure or union type, as long as the definition has the same visibility as the declaration.

Examples

```
/******************** Example 1 ********************/

typedef int WHOLE;
```

Example 1 declares WHOLE to be a synonym for **int**. Note that the keyword **const** could be used to modify WHOLE, but the type specifier **long** could not.

```
/******************** Example 2 ********************/

typedef struct club {
        char name[30];
        int size, year;
        } GROUP ;
```

Example 2 declares GROUP as a structure type with three members. Since a structure tag, `club`, is also specified, either the **typedef** name (GROUP) or the structure tag can be used in declarations.

```
/******************** Example 3 ********************/

typedef GROUP *PG;
```

Example 3 uses the previous **typedef** name to declare a pointer type. The type PG is declared as a pointer to the GROUP type, which in turn is defined as a structure type.

```
/******************** Example 4 ********************/

typedef void DRAWF(int, int);
```

Example 4 provides the type DRAWF for a function returning no value and taking two **int** arguments. This means, for example, that the declaration `DRAWF box;` is equivalent to the declaration `void box(int, int);`.

4.9 Type Names

A "type name" specifies a particular data type. In addition to ordinary variable declarations and defined-type declarations, type names are used in three other contexts: in the argument-type lists of function declarations, in type casts, and in **sizeof** operations. Argument-type lists are discussed in Section 4.5, "Function Declarations." Type casts and **sizeof** operations are discussed in sections 5.7.2 and 5.3.4, respectively.

The type names for fundamental, enumeration, structure, and union types are simply the type specifiers for those types.

A type name for a pointer, array, or function type has the following form:

type-specifier abstract-declarator

An *abstract-declarator* is a declarator without an identifier, consisting of one or more pointer, array, or function modifiers. The pointer modifier (*) always precedes the identifier in a declarator; array ([ ]) and function (( )) modifiers follow the identifier. Knowing this, you can determine where the identifier would appear in an abstract declarator and interpret the declarator accordingly. See Section 4.3.2 for information and examples of complex declarators.

Abstract declarators can be complex. Parentheses in a complex abstract declarator specify a particular interpretation, just as they do for the complex declarators in declarations.

Note

The abstract declarator consisting of a set of empty parentheses, ( ), is not allowed because it is ambiguous. It is impossible to determine whether the implied identifier belongs inside the parentheses (in which case it is an unmodified type) or before the parentheses (in which case it is a function type).

The type specifiers established by **typedef** declarations also qualify as type names.

Examples

```
/******************** Example 1 ********************/
long *
```

Example 1 gives the type name for "pointer to **long**" type.

```
/******************** Example 2 ********************/
int (*)[5]
```

```
/******************** Example 3 ********************/
int (*)(void)
```

Examples 2 and 3 show how parentheses modify complex abstract declarators. Example 2 gives the type name for a pointer to an array of five **int** values. Example 3 names a pointer to a function taking no arguments and returning an **int**.

Chapter 5
Expressions and Assignments

5.1 Introduction

This chapter describes how to form expressions and make assignments in the C language. An "expression" is a combination of operands and operators that yields ("expresses") a single value.

An "operand" is a constant or variable value that is manipulated in the expression. Each operand of an expression is also an expression, since it represents a single value. Section 5.2 describes the formats and evaluation rules for C operands.

"Operators" specify how the operand or operands of the expression are manipulated. C operators are described in Section 5.3.

In C, assignments are considered expressions because an assignment yields a value. Its value is the value being assigned. In addition to the simple-assignment operator (=), C offers complex-assignment operators that both transform and assign their operands. Assignment operators are described in Section 5.4.

When an expression is evaluated, the resulting value depends on the relative precedence of operators in the expression and on side effects, if any. The precedence of operators determines how operands are grouped for evaluation. Side effects are changes caused by the evaluation of an expression. In an expression with side effects, the evaluation of one operand can affect the value of another. With some operators, the order in which operands are evaluated also affects the result of the expression.

The value represented by each operand in an expression has a type, which may be converted to a different type in certain contexts. Type conversions occur in assignments, type casts, function calls, and operations. (Section 5.5 gives the precedence rules for C operators; side effects are discussed in Section 5.6 and type conversions in Section 5.7.)

5.2 Operands

Operands in C include constants, identifiers, strings, function calls, subscript expressions, member-selection expressions, or more complex expressions formed by combining operands with operators or enclosing operands in parentheses. Any operand that yields a constant value is called a "constant expression."

Every operand has a type. The following sections discuss the type of value each kind of operand represents. An operand can be cast from its original type to another type by means of a "type-cast" operation. A type-cast expression can also form an operand of an expression.

5.2.1 Constants

A constant operand has the value and type of the constant value it represents. A character constant has **int** type. An integer constant has **int**, **long**, **unsigned int**, or **unsigned long** type, depending on the integer's size and how the value is specified. Floating-point constants always have **double** type. String literals are considered arrays of characters and are discussed in Section 5.2.3.

5.2.2 Identifiers

An identifier names a variable or function. Every identifier has a type, which is established when the identifier is declared. The value of an identifier depends on its type, as follows:

- Identifiers of integral and floating types represent values of the corresponding type.

- An identifier of **enum** type represents one constant value among a set of constant values. The value of the identifier is the constant value. Its type is **int**, by definition of the **enum** type.

- An identifier of **struct** or **union** type represents a value of the specified **struct** or **union** type.

- An identifier declared as a pointer represents a pointer to a value of the type specified in the pointer's declaration.

- An identifier declared as an array represents a pointer whose value is the address of the first array element. The pointer addresses the type of the array elements. For example, if `series` is declared to be a 10-element integer array, the identifier `series` represents the address of the array, and the subscript expression `series[5]` refers to an integer value which is the sixth element of `series`. Subscript expressions are discussed in Section 5.2.5. The address of an array does not change during program execution, although the values of the individual elements can change. The pointer value represented by an array identifier is not a variable, so an array identifier cannot form the left-hand operand of an assignment operation.

- An identifier declared as a function represents a pointer whose value is the address of the function. The pointer addresses a function returning a value of a specified type. The address of a function does not change during program execution; only the return value varies. Thus, function identifiers cannot be left-hand operands in assignment operations.

5.2.3 Strings

Syntax

"*string*" ⟦"*string*"⟧

A string literal is a character or sequence of adjacent characters enclosed in double quotation marks. Two or more adjacent string literals separated only by white space are concatenated into a single string literal. A string literal is stored as an array of elements with **char** type, and initialized with the quoted sequence of characters. The string literal is represented by a pointer whose value is the address of the first array element. The address of the string's first element is a constant, so the value represented by a string expression is a constant.

Since string literals are effectively pointers, they can be used in contexts that allow pointer values, and they are subject to the same restrictions as pointers. However, since it is not a variable, neither the string literal nor any of its elements can be the be the left-hand operand in an assignment operation.

The last character of a string is always the null character. The null character is not visible in the string expression, but it is added as the last element when the string is stored. For example, the string `"abc"` actually has four characters rather than three.

5.2.4 Function Calls

Syntax

expression (⟦*expression-list*⟧)

A function call consists of an *expression* followed by an optional *expression-list* in parentheses, where

- The *expression* must evaluate to a function address (for example, a function identifier), and
- The *expression-list* is a list of expressions (separated by commas) whose values (the "actual arguments") are passed to the function. The *expression-list* can be empty.

A function-call expression has the value and type of the function's return value. If the function's return type is **void** (that is, the function has been declared never to return a value), the function-call expression also has **void** type. If the called function returns control without executing a **return** statement, the value of the function call expression is undefined. (See Chapter 7, "Functions," for more information about function calls.)

5.2.5 Subscript Expressions

Syntax

expression1 [*expression2*]

A subscript expression represents the value at the address that is *expression2* positions beyond *expression1*. Usually, the value represented by *expression1* is a pointer value, such as an array identifier, and *expression2* is an integral value. However, all that is required syntactically is that one of the expressions be of pointer type and the other be of integral type. Thus the integral value could be in the *expression1* position and the pointer value in the brackets in the *expression2* or "subscript" position. Whatever the order of values, *expression2* must be enclosed in brackets ([ ]).

Subscript expressions are generally used to refer to array elements, but you can apply a subscript to any pointer.

The subscript expression is evaluated by adding the integral value to the pointer value then applying the indirection operator (*) to the result. (See Section 5.3.3 for a discussion of the indirection operator.) In effect, for a one-dimensional array, the following four expressions are equivalent, assuming that a is a pointer and b is an integer:

```
a[b]
*(a + b)
*(b + a)
b[a]
```

According to the conversion rules for the addition operator (given in Section 5.3.6), the integral value is converted to an address offset by multiplying it by the length of the type addressed by the pointer.

For example, suppose the identifier `line` refers to an array of **int** values. The following procedure is used to evaluate the subscript expression `line[i]`:

1. The integer value `i` is multiplied by the length of an **int**. The converted value of `i` represents `i` **int** positions.
2. This converted value is added to the original pointer value (`line`) to yield an address that is offset `i` **int** positions from `line`.
3. The indirection operator is applied to the new address. The result is the value of the array element at that position (intuitively, `line[i]`).

Note

The subscript expression

```
line[0]
```

represents the value of the first element of `line`, since the offset from the address represented by `line` is 0. Similarly, an expression such as

```
line[5]
```

refers to the element offset five positions from `line`, or the sixth element of the array.

Multidimensional-Array References

A subscript expression can be subscripted, as follows:

expression1 [*expression2*] [*expression3*]...

Subscript expressions associate from left to right. The left-most subscript expression, *expression1*[*expression2*], is evaluated first. The address that results from adding *expression1* and *expression2* forms a pointer expression; then *expression3* is added to this pointer expression to form a new pointer expression, and so on until the last subscript expression has been added. The indirection operator (*) is applied after the last subscripted expression is evaluated, unless the final pointer value addresses an array type (see example 3 below).

Expressions with multiple subscripts refer to elements of "multidimensional arrays." A multidimensional array is an array whose elements are arrays. For example, the first element of a three-dimensional array is an array with two dimensions.

Examples

For the following examples, an array named prop is declared with three elements, each of which is a 4-by-6 array of **int** values.

```
int prop[3][4][6];
int i, *ip, (*ipp)[6];

/******************** Example 1 ********************/
i = prop[0][0][1];
```

Example 1 shows how to refer to the second individual **int** element of prop. Arrays are stored by row, so the last subscript varies the most quickly; the expression prop[0][0][2] refers to the next (third) element of the array, and so on.

```
/******************** Example 2 ********************/
i = prop[2][1][3];
```

Example 2 shows a more complex reference to an individual element of prop. The expression is evaluated as follows:

1. The first subscript, `2`, is multiplied by the size of a 4-by-6 **int** array and added to the pointer value `prop`. The result points to the third 4-by-6 array of `prop`.
2. The second subscript, `1`, is multiplied by the size of the 6-element **int** array and added to the address represented by `prop[2]` .
3. Each element of the 6-element array is an **int** value, so the final subscript, `3`, is multiplied by the size of an **int** before it is added to `prop[2][1]`. The resulting pointer addresses the fourth element of the 6-element array.
4. The indirection operator is applied to the pointer value. The result is the **int** element at that address.

```
/******************** Example 3 ********************/
ip = prop[2][1];

/******************** Example 4 ********************/
ipp = prop[2];
```

Examples 3 and 4 show cases where the indirection operator is not applied.

In Example 3, the expression `prop[2][1]` is a valid reference to the three-dimensional array `prop`; it refers to a 6-element array. Since the pointer value addresses an array, the indirection operator is not applied.

Similarly, the result of the expression `prop[2]` in Example 4 is a pointer value addressing a two-dimensional array.

5.2.6 Member-Selection Expressions

Syntax

expression.identifier
expression—>identifier

Member-selection expressions refer to members of structures and unions. A member-selection expression has the value and type of the selected member. As shown above, a member-selection expression can have one of two forms:

1. In the first form, *expression.identifier*, *expression* represents a value of **struct** or **union** type, and *identifier* names a member of the specified structure or union.
2. In the second form, *expression—>identifier*, *expression* represents a pointer to a structure or union, and *identifier* names a member of the specified structure or union.

The two forms of member-selection expressions have similar effects. In fact, an expression involving the pointer selection operator (—>) is a shorthand version of an expression using the period (.) if the expression before the period consists of the indirection operator (*) applied to a pointer value. (Section 5.3.3 discusses the indirection operator.) Therefore,

expression—>identifier

is equivalent to

(**expression*).*identifier*

when *expression* is a pointer value.

Examples

Examples 1 through 3 refer to the following structure declaration:

```
struct pair {
        int a;
        int b;
        struct pair *sp;
        } item, list[10];

/******************** Example 1 ********************/
item.sp = &item;
```

In Example 1, the address of the item structure is assigned to the sp member of the structure. This means that item contains a pointer to itself.

```
/******************** Example 2 ********************/
(item.sp)->a = 24;
```

In Example 2, the pointer expression `item.sp` is used with the pointer selection operator (—>) to assign a value to the member `a`.

```
/******************** Example 3 ********************/
list[8].b = 12;
```

Example 3 shows how to select an individual structure member from an array of structures.

5.2.7 Expressions with Operators

Expressions with operators can be“ unary,” “binary”, or “ternary” expressions. A unary expression consists of either an a unary operator (“unop”) prepended to an operand, or the **sizeof** keyword followed by an *expression*. The *expression* can be either the name of a variable or a cast expression. If *expression* is a cast expression it must be enclosed in parentheses.

unop operand
sizeof *expression*

A binary expression consists of two operands joined by a binary operator (“binop”):

operand binop operand

A ternary expression consists of three operands joined by the ternary (**?** **:**) operator:

operand **?** *operand* **:** *operand*

Sections 5.3.1 – 5.3.12 describe the operators used in unary, binary, and ternary expressions.

Expressions with operators also include assignment expressions, which use unary or binary assignment operators. The unary assignment operators are the increment (++) and decrement (—) operators; the binary assignment operators are the simple-assignment operator (=) and the compound-assignment operators (referred to as “compound-assign-ops”). Each compound-assignment operator is a combination of another binary operator with the simple-assignment operator. Assignment expressions have the following forms:

operand++

operand− −
++*operand*
− − *operand*
operand = *operand*
operand compound-assignment-op operand

Sections 5.4.1 – 5.4.4 describe the assignment operators in detail.

5.2.8 Expressions in Parentheses

You can enclose any operand in parentheses without changing the type or value of the enclosed expression. For example, in the expression

```
(10 + 5) / 5
```

the parentheses around `10 + 5` mean that the value of `10 + 5` is the left operand of the division (/) operator. The result of `(10 + 5) / 5` is 3. Without the parentheses, `10 + 5 / 5` would evaluate to 11.

Although parentheses affect the way operands are grouped in an expression, they cannot guarantee a particular order of evaluation in all cases. Exceptions resulting from "side effects" are discussed in Section 5.6.

5.2.9 Type-Cast Expressions

A type cast provides a method for explicit conversion of the type of an object in a specific situation. Type-cast expressions have the following form:

(*type-name*) *operand*

Casts can be used to convert objects of any scalar type to or from any other scalar type. Explicit type casts are constrained by the same rules that determine the effects of implicit conversions, discussed in Section 5.7.1. Additional restraints on casts may result from the actual sizes or representation of specific types on specific implementations. Representation is discussed in Chapter 4, "Declarations." For information on actual sizes of integral types and pointers, see your User's Guide.

Any object may be cast to the **void** type. However, if the *type-name* in a type-cast expression is **void**, then *operand* cannot be a **void** expression. If an object is cast to **void** type, the resulting expression cannot be assigned to any item. Similarly, a type-cast object is not an acceptable lvalue, so no assignment can be made to a type-cast object.

Section 5.7.2 discusses type-cast conversions. Section 4.9 discusses type names.

5.2.10 Constant Expressions

A constant expression is any expression that evaluates to a constant. The operands of a constant expression can be integer constants, character constants, floating type constants, enumeration constants, type casts, **sizeof** expressions, and other constant expressions. You can use operators to combine and modify operands as described in Section 5.2.7, with the following restrictions.

- You cannot use assignment operators (see Section 5.4) or the binary sequential-evaluation operator (,) in constant expressions.
- You can use the unary address-of operator (&) only in certain initializations (as described in the last paragraph of Section 5.2.10).

Constant expressions used in preprocessor directives are subject to additional restrictions. Consequently, they are known as *restricted-constant-expressions.* A *restricted-constant-expression* cannot contain **sizeof** expressions, enumeration constants, type casts to any type, or floating type constants. It can, however, contain the special constant expression **defined(***identifier***)**. (See Section 8.2.1, "The #define Directive," for more information about this expression.)

Constant expressions involving floating constants, casts to nonarithmetic types, and address-of expressions can only appear in initializers. The unary address-of operator (&) can only be applied variables with fundamental, structure, or union types that are declared at the external level, or to subscripted array references. In these expressions, a constant expression that does not include the address-of operator can be added to or subtracted from the address expression.

5.2.11 Sequence Points

Expressions involving assignment, unary "increment," unary "decrement," or calling a function may have consequences incidental to their evaluation called "side effects." When a "sequence point" is reached, everything preceding the sequence point, including any side effects, is guaranteed to have been evaluated before evaluation begins on anything following the sequence point.

Certain operators act as sequence points, including the following:

- The logical-AND operator (**&&**)
- The logical-OR operator (**||**)
- The ternary opertor (**?:**)
- The sequential-evaluation operator (**,**)
- The function call operator (that is, the parenthese following a function name)
- The unary plus operator (**+**), though not yet implemented, is defined as a sequence point

Other sequence points include the end of a full expression (that is, an expression that is not part of another expression); any initializer; an expression in an expression statement; the control expressions in selection statements (**if** or **switch**) and iteration statements (**do**, **while**, or **for**); the expression in a **return** statement.

Section 5.6 discusses side effects in more detail.

5.3 Operators

C operators take one operand (unary operators), two operands (binary operators), or three operands (the ternary operator). Assignment operators include both unary or binary operators; Section 5.4 describes the assignment operators.

Unary operators appear before their operand and associate from right to left. C includes the following unary operators:

Symbol	**Name**
− ~ !	Negation and complement operators
* &	Indirection and address-of operators
sizeof	Size operator

Binary operators associate from left to right. C provides the following binary operators:

Symbol	Name
* / %	Multiplicative operators
+ −	Additive operators
<< >>	Shift operators
< > <= >= == !=	Relational operators
& | ^	Bitwise operators
&& ||	Logical operators
,	Sequential-evaluation operator

C has one ternary operator, the conditional operator (**? :**). It associates from right to left.

5.3.1 Usual Arithmetic Conversions

Most C operators perform type conversions to bring the operands of an expression to a common type or to extend short values to the integer size used in machine operations. The conversions performed by C operators depend on the specific operator and the type of the operand or operands. However, many operators perform similar conversions on operands of integral and floating types. These conversions are known as "arithmetic" conversions because they apply to the types of values ordinarily used in arithmetic.

The arithmetic conversions summarized below are called the "usual arithmetic conversions." The discussion of each operator in the following sections specifies whether or not the operator performs the usual arithmetic conversions. It also specifies the additional conversions, if any, the operator performs. This is not a precedence order. It is an outline of an algorithm that is applied to each binary operator in the expression.

Section 5.7 outlines the specific path of each type of conversion. In determining the "usual arithmetic conversions" the following algorithm is applied to each binary operation in the expression:

1. Any operands of **float** type are converted to **double** type.
2. If one operand has **long double** type, the other operand is converted to **long double** type.
3. If one operand has **double** type, the other operand is converted to **double** type.
4. Any operands of **char** or **short** type are converted to **int** type.
5. Any operands of **unsigned char** or **unsigned short** type are converted to **unsigned int** type.
6. If one operand is of **unsigned long** type, the other operand is converted to **unsigned long** type.
7. If one operand is of **long** type, the other operand is converted to **long** type.
8. If one operand is of **unsigned int** type, the other operand is converted to **unsigned int** type.

The following example illustrates the application of the preceding algorithm:

```
long l;
unsigned char uc;
int i;
f( l + uc * i);
```

The preceding example would be converted as follows:

1. `uc` is converted to an **unsigned int** (step 4).
2. `i` is converted to an **unsigned int** (step 7) The multiplication is performed and the result is an **unsigned int**.
3. `uc * i` is converted to a **long**(step**6).**

The addition is performed and the result is type **long**.

5.3.2 Complement Operators

Arithmetic Negation (–)

The arithmetic-negation operator (–) produces the negative (two's complement) of its operand. The operand must be an integral or floating value. This operator performs the usual arithmetic conversions.

Bitwise Complement (~)

The bitwise-complement operator (~) produces the bitwise complement of its operand. The operand must be of integral type. This operator performs usual arithmetic conversions; the result has the type of the operand after conversion.

Logical-NOT (!)

The logical-NOT operator (!) produces the value 0 if its operand is true (nonzero) and the value 1 if its operand is false (0). The result has **int** type. The operand must be an integral, floating, or pointer value.

Examples

```
/******************** Example 1 *******************/
short x = 987;
    x = -x;
```

In Example 1, the new value of x is the negative of 987, or —987.

```
/******************** Example 2 *******************/
unsigned short y = 0xaaaa;
    y = ~y;
```

In Example 2, the new value assigned to y is the one's complement of the unsigned value 0xaaaa, or 0x5555.

```
/******************** Example 3 *******************/
if ( !(x < y));
```

In Example 3, if x is greater than or equal to y, the result of the expression is 1 (true). If x is less than y, the result is 0 (false).

5.3.3 Indirection and Address-of Operators

Indirection (*)

The indirection operator (*) accesses a value indirectly, through a pointer. The operand must be a pointer value. The result of the operation is the value that the operand points to; that is, the value at the address specified by the operand. The result type is the type that the operand addresses. If the pointer value is invalid, the result is unpredictable. The specific conditions that invalidate a pointer value are implementation-defined, and the following list includes some of the most common:

- A pointer that is a null pointer
- A pointer that specifies the address of a local item that is not active at the time of the reference
- A pointer to an address that is inappropriately aligned for the type of the object pointed to
- A pointer to an address not used by the executing program

Address-of (&)

The address-of operator (&) gives the address of its operand. The operand can be any value that can appear as the left-hand value of an assignment operation. A function designator or array name can also be the operand of the address-of operator, although in these cases the operator is superfluous since function designators and array names are addresses. (Assignment operations are discussed in Section 5.4.) The result of the address operation is a pointer to the operand. The type addressed by the pointer is the type of the operand.

You cannot apply the address-of operator to a bit-field member of a structure (described in Section 4.4.3) or to an identifier declared with the **register** storage-class specifier (described in Section 4.6).

Examples

Examples 1 through 4 use the following declarations:

```
int *pa, x;
int a[20];
```

```
double d;

/******************** Example 1 ********************/
pa = &a[5];
```

In Example 1, the address-of operator (&) takes the address of the sixth element of the array a. The result is stored in the pointer variable pa.

```
/******************** Example 2 ********************/
x = *pa;
```

The indirection operator (*) is used in Example 2 to access the **int** value at the address stored in pa. The value is assigned to the integer variable x.

```
/******************** Example 3 ********************/
if (x == *&x)
    printf("True\n");
```

In Example 3, the word True would be printed. This example demonstrates that the result of applying the indirection operator to the address of x is the same as x.

```
/******************** Example 4 ********************/
d = *(double *)(&x);
```

Example 4 shows a useful application of the rule shown in Example 3. First the address of x is converted by a type cast to a pointer to a **double** type; then the indirection operator is applied to give a result of type **double**.

```
/******************** Example 5 ********************/
int foo() ;

int *pfoo = foo;
int *pfo = &foo;
```

In Example 5, the function foo is declared, and then two pointers to foo are declared and initialized. The first pointer pfoo is initialized using only the name of the function, while the second, pfo uses the address-of operator in the initialization. The initializations are equivalent.

5.3.4 The sizeof Operator

The **sizeof** operator gives the amount of storage, in bytes, associated with an identifier or a type. This operator allows you to avoid specifying machine-dependent data sizes in your programs.

A **sizeof** expression has the form

sizeof*expression*

where *expression* is either an identifier or a type-cast expression (that is, a type specifier enclosed in parentheses). If *expression* is a type-cast expression, it cannot be **void**. If it is an identifier, it cannot represent a bit-field object of a function designator.

When you apply the **sizeof** operator to an array identifier, the result is the size of the entire array rather than the size of the pointer represented by the array identifier.

When you apply the **sizeof** operator to a structure or union type name, or to an identifier of structure or union type, the result is the actual size of the structure or union. This size may include internal and trailing padding used to align the members of the structure or union on memory boundaries. Thus, the result may not correspond to the size calculated by adding up the storage requirements of the individual members.

Examples

```
/******************** Example 1 ********************/
buffer = calloc(100, sizeof (int) );
```

Example 1 uses the **sizeof** operator to pass the size of an **int**, which varies among machines, as an argument to a function named `calloc`. The value returned by the function is stored in `buffer`.

```
/******************** Example 2 ********************/
static char *strings[] ={
                          "this is string one",
                          "this is string two",
                          "this is string three",
                        };
const int string_no = (sizeof strings)/(sizeof strings[0]);
```

In Example 2 `strings` is an array of pointers to **char**. The number of pointers is the number of elements in the array, but is not specified. It is easy to determine the number of pointers by using the **sizeof** operator to calculate the number of elements in the array. The **const** integer value `string_no` is initialized to this number. Because it is a **const**, `string_no` cannot be modified.

5.3.5 Multiplicative Operators

The multiplicative operators perform multiplication (*), division (/), and remainder (%) operations. The operands of the remainder operator (%) must be integral. The multiplication (*) and division (/) operators can take integral or floating type operands; the types of the operands can be different.

The multiplicative operators perform the usual arithmetic conversions on the operands. The type of the result is the type of the operands after conversion.

Note

Since the conversions performed by the multiplicative operators do not provide for overflow or underflow conditions, information may be lost if the result of a multiplicative operation cannot be represented in the type of the operands after conversion.

Multiplication (*)

The multiplication operator (*) causes its two operands to be multiplied.

Division (/)

The division operator (/) causes the first operand to be divided by the second. If two integer operands are divided and the result is not an integer, it is truncated according to the following rules:

- If both operands are positive or unsigned, the result is truncated toward 0.
- If either operand is negative, the direction of truncation of the result (either toward 0 or away from 0), is defined by the implementation. For more information, see your User's Guide.

The result of division by 0 is undefined.

Remainder (%)

The result of the remainder operator (%) is the remainder when the first operand is divided by the second. If either or both operands are positive or unsigned, the result is positive. If either operand is negative the sign of the result is defined by the implementation. (See your User's Guide for more information.) If the right operand is zero, the result is undefined.

Examples

The following declarations are used for all of the following examples:

```
int i = 10, j = 3, n;
double x = 2.0, y;

/******************** Example 1 ********************/
y = x * i;
```

In Example 1, `x` is multiplied by `i` to give the value 20.0. The result has **double** type.

```
/******************** Example 2 ********************/
n = i / j;
```

In Example 2, 10 is divided by 3. The result is truncated toward 0, yielding the integer value 3.

```
/******************** Example 3 ********************/
n = i % j;
```

In Example 3, `n` is assigned the integer remainder, 1, when 10 is divided by 3.

5.3.6 Additive Operators

The additive operators perform addition (+) and subtraction (–). The operands can be integral or floating values. Some additive operations can also be performed on pointer values, as outlined under the discussion of each operator.

The additive operators perform the usual arithmetic conversions on integral and floating operands. The type of the result is the type of the operands after conversion. Since the conversions performed by the additive operators do not provide for overflow or underflow conditions, information may be lost if the result of an additive operation cannot be represented in the type of the operands after conversion.

Addition (+)

The addition operator (+) causes its two operands to be added. Both operands can have integral or floating types, or one operand can be a pointer and the other an integer.

When an integer is added to a pointer, the integer value (*i*) is converted by multiplying it by the size of the value that the pointer addresses. After conversion, the integer value represents *i* memory positions, where each position has the length specified by the pointer type. When the converted integer value is added to the pointer value, the result is a new pointer value representing the address *i* positions from the original address. The new pointer value addresses a value of the same type as the original pointer value.

Subtraction (–)

The subtraction operator (–) subtracts the second operand from the first. The following combinations of operands can be used with this operator:

- Both operands integral or floating type values
- Both operands pointer values to the same type
- The first operand a pointer value and the second operand an integer

When two pointers are subtracted, the difference is converted to a signed integral value by dividing the difference by the size of a value of the type that the pointers address. The size of the integral value is defined by the implementation. (See your User's Guide for more information.) The result represents the number of memory positions of that type between the two addresses. The result is only guaranteed to be meaningful for two elements of the same array, as discussed under "Pointer Arithmetic" later in this section.

When an integer value is subtracted from a pointer value, the subtraction operator converts the integer value (*i*) by multiplying it by the size of the value that the pointer addresses. After conversion, the integer value represents *i* memory positions, where each position has the length specified by the pointer type. When the converted integer value is subtracted from the pointer value, the result is the memory address *i* positions before the original address. The new pointer points to a value of the type addressed by the original pointer value.

Pointer Arithmetic

Additive operations involving a pointer and an integer give meaningful results only if the pointer operand addresses an array member and the integer value produces an offset within the bounds of the same array. When the integer value is converted to an address offset, the compiler assumes that only memory positions of the same size lie between the original address and the address plus the offset.

This assumption is valid for array members. By definition, an array is a series of values of the same type; its elements reside in contiguous memory locations. However, storage for any types except array elements is not guaranteed to be completely filled. That is, blanks may appear between memory positions, even positions of the same type. Therefore, the results of adding to or subtracting from the addresses of any values but array elements are undefined.

Similarly, when two pointer values are subtracted, the conversion assumes that only values of the same type, with no blanks, lie between the addresses given by the operands.

On machines with segmented architecture (such as the 8086/8088), additive operations between pointer and integer values may not be valid in some cases. For example, an operation may result in an address that is outside the bounds of an array. See your User's Guide discussion of memory models for more information.

Examples

The following declarations are used for both examples:

```
int i = 4, j;
float x[10];
float *px;

/******************** Example 1 ********************/
px = &x[4] + i;  /* equivalent to px = &x[4=i]; */
```

In Example 1, the value of `i` is multiplied by the length of a **float** and added to `&x[4]`. The resulting pointer value is the address of `x[8]`.

```
/******************** Example 2 ********************/
j = &x[i] - &x[i-2];
```

In Example 2, the address of the third element of `x` (given by `x[i-2]`) is subtracted from the address of the fifth element of `x` (given by `x[i]`). The difference is divided by the length of a **float**; the result is the integer value 2.

5.3.7 Shift Operators

The shift operators shift their first operand left (<<) or right (>>) by the number of positions the second operand specifies. Both operands must be integral values. These operators perform the usual arithmetic conversions; the type of the result is the type of the left operand after conversion.

For leftward shifts, the vacated right bits are set to 0. For rightward shifts, the vacated left bits are filled based on the type of the first operand after conversion. If the type is **unsigned**, they are set to 0. Otherwise, they are filled with copies of the sign bit.

The result of a shift operation is undefined if the second operand is negative.

Since the conversions performed by the shift operators do not provide for overflow or underflow conditions, information may be lost if the result of a shift operation cannot be represented in the type of the first operand after conversion.

Example

```
unsigned int x, y, z;

x = 0x00aa;
y = 0x5500;

z = (x << 8) + (y >> 8);
```

In this example, x is shifted left eight positions and y is shifted right eight positions. The shifted values are added, giving 0xaa55, and assigned to z.

5.3.8 Relational Operators

The binary relational operators compare their first operand to their second operand to test the validity of the specified relationship. The result of a relational expression is 1 if the tested relationship is true and 0 if it is false. The type of the result is **int**.

The relational operators test the following relationships:

Operator **Relationship Tested**

<	First operand less than second operand
>	First operand greater than second operand
<=	First operand less than or equal to second operand
>=	First operand greater than or equal to second operand
==	First operand equal to second operand
!=	First operand not equal to second operand

The operands can have integral, floating, or pointer type. The types of the operands can be different. Relational operators perform the usual arithmetic conversions on integral and floating type operands. In addition, you can use the following combinations of operand types with relational operators:

- Both operands of any relational operator can be pointers to the same type. For the equality (==) and inequality (!=) operators, the result of the comparison indicates whether or not the two pointers address the same memory location. For the other relational operators (<, >, <=, and >=), the result of the comparison indicates the relative position of two memory addresses.

  Since the address of a given value is arbitrary, comparisons between the addresses of two unrelated values are generally meaningless. However, comparisons between the addresses of different elements of the same array can be useful, since array elements are guaranteed to be stored in order from the first element to the last. The address of the first array element is "less than" the address of the last element.

- A pointer value can be compared to the constant value 0 for equality (==) or inequality (!=). A pointer with a value of 0, called a "null" pointer, does not point to a memory location.

Examples

```
/********************* Example 1 ********************/
int x = 0, y = 0;

x < y
```

Because x and y are equal, the expression in Example 1 yields the value 0.

```
/********************* Example 2 ********************/
char array[10] ;
char *p ;

  for (p=array; p<&array[10]; p++)
       *p = '\0' ;
```

The fragment in Example 2 initializes each element of `array` to a null character constant.

```
/******************** Example 3 *********************/
enum color {red, white, green} col;
        .
        .
        .
        if (col == red)
        .
        .
        .
```

Example 3 declares an enumeration variable named `col` with the tag `color`. At any time, the variable may contain an integer value of 0, 1, or 2, representing one of the elements of the enumeration set `color`: the colors red, white and green respectively. If `col` contains 0 when the **if** statement is executed, any statements depending on the **if** will be executed.

5.3.9 Bitwise Operators

The bitwise operators perform bitwise-AND (&), inclusive-OR ( ) and exclusive-OR (^) operations. The operands of bitwise operators must have integral types, but their types can be different. These operators perform the usual arithmetic conversions; the type of the result is the type of the operands after conversion.

Bitwise AND (&)

The bitwise-AND (&) operator compares each bit of its first operand to the corresponding bit of its second operand. If both bits are 1, the corresponding result bit is set to 1; otherwise, the corresponding result bit is set to 0.

Bitwise Inclusive OR (¦)

The bitwise-inclusive-OR (¦) operator compares each bit of its first operand to the corresponding bit of its second operand. If either bits is 1, the corresponding result bit is set to 1. Otherwise the corresponding result bit is set to 0.

Bitwise Exclusive OR (^)

The bitwise-exclusive-OR (^) operator compares each bit of its first operand to the corresponding bit of its second operand. If one bit is 0 and the other bit is 1, the corresponding result bit is set to 1; otherwise, the corresponding result bit is set to 0.

Examples

```
short i = 0xab00;
short j = 0xabcd;
short n;

/******************** Example 1 ********************/
n = i & j;

/******************** Example 2 ********************/
n = i | j;

/******************** Example 3 ********************/
n = i ^ j;
```

The result assigned to `n` in the first example is the same as `i` (AB00 hexadecimal). The bitwise inclusive OR in Example 2 results in the value ABCD (hexadecimal), while the bitwise exclusive OR in example 3 produces CD (hexadecimal).

5.3.10 Logical Operators

The logical operators perform logical-AND (&&) and logical-OR (||) operations. The operands of the logical operators must have integral, floating, or pointer type. The types of the operands can be different.

The operands of logical-AND and logical-OR expressions are evaluated from left to right. If the value of the first operand is sufficient to determine the result of the operation, the second operand is not evaluated. There is a sequence point after the first operand.

Logical operators do not perform the standard arithmetic conversions. Instead, they evaluate each operand in terms of its equivalence to 0.

The result of a logical operation is either 0 or 1. The type of the result is **int**.

Logical AND (&&)

The logical-AND operator (&&) produces the value 1 if both operands have nonzero values. If either operand is equal to 0, the result is 0. If the first operand of a logical-AND operation is equal to 0, the second operand is not evaluated.

Logical OR (||)

The logical-OR operator (||) performs an inclusive-OR operation on its operands. The result is 0 if both operands have 0 values. If either operand has a nonzero value, the result is 1. If the first operand of a logical-OR operation has a nonzero value, the second operand is not evaluated.

Examples

```
int w, x, y, z;

/******************** Example 1 ********************/
if (x < y && y < z)
        printf ("x is less than z\n");
```

In Example 1, the `printf` function is called to print a message if `x` is less than `y` and `y` is less than `z`. If `x` is greater than `y`, the second operand (`y < z`) is not evaluated and nothing is printed. Note that this could cause problems in cases where the second operand contains side effects.

```
/******************** Example 2 ********************/
        printf ("%d" , (x==w || x==y || x==z));
```

In Example 2, if `x` is equal to either `w`, `y`, or `z`, the second argument to the `printf` function evaluates to true and the value 1 is printed. Otherwise it evaluates to false and the value 0 is printed. As soon as one of the conditions evaluates to true, evaluation ceases.

5.3.11 Sequential-Evaluation Operator

The sequential-evaluation operator (,) evaluates its two operands sequentially from left to right. There is a seqence point after the first operand. The result of the operation has the same value and type as the right operand. The operands can be of any types. The sequential-evaluation operator does not perform type conversions.

The sequential-evaluation operator, also called the "comma" operator, is typically used to evaluate two or more expressions in contexts where only one expression is allowed.

Note that commas may be used as a separators in some contexts. You must be careful not to confuse the use of the comma as a separator with its use as an operator; the two uses are completely different.

Examples

```
/******************** Example 1 ********************/

for ( i = j = 1; i + j < 20; i += i, j--);
```

In Example 1, each operand of the **for** statement's third expression is evaluated independently. The left operand, `i += i`, is evaluated first; then the right operand, `j--`, is evaluated.

```
/******************** Example 2 ********************/

func_one(x, y + 2, z);
```

```
func_two((x--, y + 2), z);
```

In the function call to `func_one`, three arguments, separated by commas, are passed: `x`, `y + 2`, and `z`.

In the function call to `func_two`, parentheses force the compiler to interpret the first comma as the sequential-evaluation operator. This function call passes two arguments to `func_two`. The first argument is the result of the sequential-evaluation operation `(x--, y + 2)`, which has the value and type of the expression `y + 2`; the second argument is `z`.

5.3.12 Conditional Operator

C has one ternary operator: the conditional operator (**? :**). It has the following form:

operand1 **?** *operand2* **:** *operand3*

The expression *operand1* must have integral, floating, or pointer type. It is evaluated in terms of its equivalence to 0: There is a sequence point following *operand1*.

- If *operand1* does not evaluate to 0, *operand2* is evaluated, and the result of the expression is the value of *operand2*.
- If *operand1* evaluates to 0, *operand3* is evaluated, and the result of the expression is the value of *operand3*.

Note that either *operand2* or *operand3* is evaluated, but not both.

The type of the result of a conditional operation depends on the type of *operand2* or *operand3*, as follows:

- If *operand2* or *operand3* has integral or floating type (their types can be different), the operator performs the usual arithmetic conversions. The type of the result is the type of the operands after conversion.
- If both *operand2* and *operand3* have the same structure, union, or pointer type, the type of the result is the same structure, union, or pointer type.
- If both operands have type **void**, the result has type **void**.
- If either operand is a pointer to an object of any type, and the other operand is a pointer to **void**, the pointer to the object is converted to a pointer to **void** and the result is a pointer to **void**.

- If either *operand2* or *operand3* is a pointer and the other operand is a constant expression with the value 0, the type of the result is the pointer type.

Examples

```
/******************** Example 1 ********************/
j = (i < 0) ? (-i) : (i);
```

Example 1 assigns the absolute value of `i` to `j`. If `i` is less than 0, `-i` is assigned to `j`. If `i` is greater than or equal to 0, `i` is assigned to `j`.

```
/******************** Example 2 ********************/
void f1(void) ;
void f2(void) ;
int x ;
int y ;
    .
    .
    .
(x==y) ? (f1()) : (f2()) ;
```

In Example 2 two functions `f1` and `f2` and two variables `x` and `y` are declared. Later in the program, if the two variables have the same value, the function `f1` is called. Otherwise `f2` is called.

5.4 Assignment Operators

The assignment operators in C can both transform and assign values in a single operation. Using a compound-assignment operator to replace two separate operations can make your programs smaller and more efficient.

C provides the following assignment operators:

Operator	Operation Performed
++	Unary increment
--	Unary decrement
=	Simple assignment
*=	Multiplication assignment

/=	Division assignment
%=	Remainder assignment
+=	Addition assignment
−=	Subtraction assignment
<<=	Left-shift assignment
>>=	Right-shift assignment
&=	Bitwise-AND assignment
|=	Bitwise-inclusive-OR assignment
^=	Bitwise-exclusive-OR assignment

In assignment, the type of the right-hand value is converted to the type of the left-hand value. The specific conversion path, which depends on the two types, is outlined in detail in Section 5.7.

5.4.1 Lvalue Expressions

An assignment operation assigns the value of the right-hand operand to the storage location named by the left-hand operand. Therefore, the left-hand operand of an assignment operation (or the single operand of a unary assignment expression) must be an expression that refers to a modifiable memory location.

Expressions that refer to memory locations are called "lvalue" expressions. Expressions referring to modifiable locations are modifiable lvalues. One example of a modifiable lvalue expression is a variable name declared without the **const** specifier. The name of the variable denotes a storage location, while the value of the variable is the value stored at that location.

The following C expressions may be lvalue expressions:

- An identifier of integral, floating, pointer, structure, or union type
- A subscript ([ ]) expression that does not evaluate to an array or a function
- A member-selection expression (−> and .), if the selected member is one of the aforementioned expressions
- A unary-indirection (*) expression that does not refer to an array or function

- An lvalue expression in parentheses.
- A **const** object is a nonmodifiable lvalue.

Note

When extensions to the ANSI C standard are enabled, a type cast to a pointer type is an lvalue expression, as long as the size of the object does not change. See your User's Guide for information on enabling and disabling the Microsoft extensions.

5.4.2 Unary Increment and Decrement

The unary assignment operators (++ and −−) increment and decrement their operand, respectively. The operand must have integral, floating, or pointer type and must be a modifiable (non-**const**) lvalue expression.

An operand of integral or floating type is incremented or decremented by the integer value 1. The result type is the same as the operand type. An operand of pointer type is incremented or decremented by the size of the object it addresses. An incremented pointer points to the next object; a decremented pointer points to the previous object.

An increment (++) or decrement (−−) operator can appear either before or after its operand, with the following results:

- When the operator appears before its operand, the operand is incremented or decremented and its new value is the result of the expression.
- When the operator appears after its operand, the immediate result of the expression is the value of the operand *before* it is incremented or decremented. After that result is applied in context, the operand is incremented or decremented.

Examples

```
/******************** Example 1 ********************/

if (pos++ > 0)
        *p++ = *q++;
```

In Example 1, the variable pos is compared to 0, then incremented. If pos was positive before being incremented, the next statement is executed. First the value of q is assigned to p. Then, q and p are incremented.

```
/******************** Example 2 ********************/

if (line[--i] != '\n')
        return;
```

In Example 2, the variable i is decremented before it is used as a subscript to line.

5.4.3 Simple Assignment

The simple-assignment operator (=) assigns its right operand its left operand. The conversion rules for assignment apply (see Section 5.7.1).

Example

```
double x;
int y;

x = y;
```

In this example, the value of y is converted to **double** type and assigned to x.

5.4.4 Compound Assignment

The compound-assignment operators combine the simple-assignment operator with another binary operator. Compound-assignment operators perform the operation specified by the additional operator, then assign the result to the left operand. For example, a compound-assignment expression such as

expression1 += *expression2*

can be understood as

expression1 = *expression1* + *expression2*

However, the compound-assignment expression is not equivalent to the expanded version because the compound-assignment expression evaluates *expression1* only once, while the expanded version evaluates *expression1* twice: in the addition operation and in the assignment operation.

The operands of a compound-assignment operator must be of integral or floating type. Each compound-assignment operator performs the conversions that the corresponding binary operator performs and restricts the types of its operands accordingly. The addition-assignment (+=) and subtraction-assignment (−=) operators may also have a left operand of pointer type, in which case the right-hand operand must be of integral type. The result of a compound-assignment operation has the value and type of the left operand.

Example

```
#define MASK   0xff00

n &= MASK;
```

In this example a bitwise-inclusive-AND operation is performed on `n` and `MASK`, and the result is assigned to `n`. The manifest constant `MASK` is defined with a **#define** preprocessor directive; this directive is discussed in Section 8.2.1.

5.5 Precedence and Order of Evaluation

The precedence and associativity of C operators affect the grouping and evaluation of operands in expressions. 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.

Table 5.1 summarizes the precedence and associativity of C operators, listing them in order of precedence from highest to lowest. Where several operators appear together in a line or large brace, they have equal precedence and are evaluated according to their associativity.

Table 5.1

Precedence and Associativity of C Operators

Symbol[a]	Type of Operation	Associativity
`() [] . ->`	Expression	Left to right
`- ~ ! * &` `++ --` **sizeof** *casts*	Unary[b]	Right to left
`* / %`	Multiplicative	Left to right
`+ -`	Additive	Left to right
`<< >>`	Shift	Left to right
`< > <= >=`	Relational (inequality)	Left to right
`== !=`	Relational (equality)	Left to right
`&`	Bitwise-AND	Left to right
`^`	Bitwise-exclusive-OR	Left to right
`|`	Bitwise-inclusive-OR	Left to right
`&&`	Logical AND	Left to right
`||`	Logical OR	Left to right
`? :`	Conditional	Right to left
`= *= /= %=` `+= -= <<= >>=` `&= |= ^=`	Simple and compound assignment[c]	Right to left
`,`	Sequential evaluation	Left to right

[a] Operators are listed in descending order of precedence. If several operators appear in the same line or in a large brace, they have equal precedence.

[b] All unary operators have equal precedence.

[c] All simple and compound-assignment operators have equal precedence.

As Table 5.1 shows, operands consisting of a constant, an identifier, a string, a function call, a subscript expression, a member-selection expression, or a parenthetical expression have the highest precedence and associate from left to right. Type-cast conversions have the same precedence and associativity as the unary operators.

An expression can contain several operators with equal precedence. When several such operators appear at the same level in an expression, evaluation proceeds according to the associativity of the operator, either from right to left or from left to right. The direction of evaluation does not affect the results of expressions that include more than one multiplicaion (*), addition (+), or binary-bitwise (& ¦ ^) operator at the same level. The compiler is free to evaluate such expressions in any order, even when parentheses in the expression appear to specify a particular order.

Important

Only the sequential-evaluation (,), logical-AND (&&), logical-OR ( ), ternary (?:) and function-call operators constitute sequence points, and therefore guarantee a particular order of evaluation for their operands. The function-call operator is the parentheses following the function identifier. The sequential-evaluation operator (,) is guaranteed to evaluate its operands from left to right. (Note that the comma separating arguments in a function call is not the same as the sequential-evaluation operator and does not provide any such guarantee.) The unary plus operator (+), is not yet implemented, but is defined by the ANSI C standard as a sequence point. Sequence points are discussed in Section 5.2.11.

Logical operators also guarantee evaluation of their operands from left to right. However, they evaluate the smallest number of operands needed to determine the result of the expression. Thus, some operands of the expression may not be evaluated. For example, in the expression x && y++, the second operand, y++, is evaluated only if x is true (nonzero). Thus, y is not incremented if x is false (0).

Examples

The following list shows the default groupings for several sample expressions:

Expression	Default Grouping
`a & b || c`	`(a & b) || c`
`a = b || c`	`a = (b || c)`
`q && r || s--`	`(q && r) || s--`

In the first expression, the bitwise-AND operator (**&**) has higher precedence than the logical-OR operator (**¦¦**), so `a & b` forms the first operand of the logical-OR operation.

In the second expression, the logical-OR operator ( )precedence than the simple-assignment operator (**=**), so `b || c` is grouped as the right-hand operand in the assignment. Note that the value assigned to `a` is either 0 or 1.

The third expression shows a correctly formed expression that may produce an unexpected result. The logical-AND operator (**&&**) has higher precedence than the logical-OR operator (**¦¦**), so `q && r` is grouped as an operand. Since the logical operators guarantee evaluation of operands from left to right, `q && r` is evaluated before `s--`. However, if `q && r` evaluates to a nonzero value, `s--` is not evaluated, and `s` is not decremented. To correct this problem, `s--` should appear as the first operand of the expression, or `s` should be decremented in a separate operation.

The following expression is illegal and produces a program error:

Illegal Expression	Default Grouping
`p == 0 ? p += 1: p += 2`	`(p == 0 ? p += 1 : p) += 2`

In this expression, the equality operator (**= =**) has the highest precedence, so `p == 0` is grouped as an operand. The ternary operator (**? :**) has the next-highest precedence. Its first operand is `p == 0`, and its second operand is `p += 1`. However, the last operand of the ternary operator is considered to be `p` rather than `p += 2`, since this occurrence of `p` binds more closely to the ternary operator than it does to the compound-assignment operator. A syntax error occurs because `+= 2` does not have a left-hand operand. You should use parentheses to prevent errors of this kind and produce more readable code. For example, you could use parentheses as shown below to correct and clarify the preceding example:

```
(p == 0) ? (p += 1) : (p += 2)
```

5.6 Side Effects

"Side effects" occur whenever the value of a variable is changed by expression evaluation. All assignment operations have side effects. Function calls may also have side effects if they change the value of an externally visible item, either by direct assignment or by indirect assignment through a pointer.

The order of evaluation of expressions is defined by the specific implementation, except when the languageguarantees a particular order of evaluation (as outlined in Section 5.5).

For example, side effects occur in the following function call:

```
add (i + 1, i = j + 2)
```

The arguments of a function call can be evaluated in any order. The expression `i + 1` may be evaluated before `i = j + 2`, or `i = j + 2`, may be evaluated before `i + 1`. The result is different in each case.

Since unary increment and decrement operations involve assignments, such operations can cause side effects, as shown in the following example:

```
d = 0;
a = b++ = c++ = d++;
```

In this example, the value of `a` is unpredictable. The value of `d` (initially 0) could be assigned to `c`, then to `b`, and then to `a` before any of the variables are incremented. In this case, `a` would be equal to 0.

A second way to evaluate this expression begins by evaluating the operand `c++ = d++`. The value of `d` (initially 0) is assigned to `c`, and then both `d` and `c` are incremented. Next, the value of `c`, now 1, is assigned to `b` and `b` is incremented. Finally, the incremented value of `b` is assigned to `a`; in this case, the final value of `a` is 2.

Since C does not define the order of evaluation of side effects, both of these evaluation methods are correct and either may be implemented. To make sure that your code is portable and clear, avoid statements that depend on a particular order of evaluation for side effects.

5.7 Type Conversions

Type conversions are performed in the following cases:

- When a value of one type is assigned to a variable of a different type
- When a value of one type is explicitly cast to a different type
- When an operator converts the type of its operand or operands before performing an operation
- When a value is passed as an argument to a function.

Sections 5.7.1.1 through 5.7.1.5 outline the rules for each kind of conversion.

5.7.1 Assignment Conversions

In assignment operations, the type of the value being assigned is converted to the type of the variable that receives the assignment. C allows conversions by assignment between integral and floating types, even if information is lost in the conversion. The conversion methods depend on the types involved in the assignment, as described in Section 5.3.1, and Sections 5.7.1.1 – 5.7.1.5.

5.7.1.1 Conversions from Signed Integral Types

A signed integer is converted to a shorter signed integer by truncating the high-order bits, and to a longer signed integer by sign extension.

When a signed integer is converted to an unsigned integer, the signed integer is converted to the size of the unsigned integer, and the result is interpreted as an unsigned value.

No information is lost when a signed integer is converted to a floating value, except that some precision may be lost when a **long int** or **unsigned long int** value is converted to a **float** value.

Table 5.2 summarizes conversions from signed integral types. This table assumes that the **char** type is signed by default. If you use a compile-time option to change the default for the **char** type to unsigned, the conversions given in Table 5.3 for the **unsigned char** type apply instead of the conversions in Table 5.2.

Table 5.2

Conversions from Signed Integral Types

From	To	Method
char[a]	**short**	Sign extend
char	**long**	Sign extend
char	**unsigned char**	Preserve pattern; high-order bit loses function as sign bit
char	**unsigned short**	Sign extend to **short**; convert **short** to **unsigned short**
char	**unsigned long**	Sign extend to **long**; convert **long** to **unsigned long**
char	**float**	Sign extend to **long**; convert **long** to **float**
char	**double**	Sign extend to **long**; convert **long** to **double**
short	**char**	Preserve low-order byte
short	**long**	Sign extend
short	**unsigned char**	Preserve low-order byte
short	**unsigned short**	Preserve bit pattern; high-order bit loses function as sign bit
short	**unsigned long**	Sign extend to **long**; convert **long** to **unsigned long**
short	**float**	Sign extend to **long**; convert **long** to **float**
short	**double**	Sign extend to **long**; convert **long** to **double**
long	**char**	Preserve low-order byte
long	**short**	Preserve low-order word
long	**unsigned char**	Preserve low-order byte
long	**unsigned short**	Preserve low-order word
long	**unsigned long**	Preserve bit pattern; high-order bit loses function as sign bit
long	**float**	Represent as **float**; if **long** cannot be represented exactly, some precision is lost precision occurs
long	**double**	Represent as **double**; if **long** cannot be represented exactly as a **double**, some precision is lost

[a] All **char** entries assume that the **char** type is signed by default.

Note

The **int** type is equivalent to either the **short** type or the **long** type, depending on the implementation. Conversion of an **int** value proceeds the same as for a **short** or a **long**, whichever is appropriate.

5.7.1.2 Conversions from Unsigned Integral Types

An unsigned integer is converted to a shorter unsigned or signed integer by truncating the high-order bits, or to a longer unsigned or signed integer by zero-extending.

When an unsigned integer is converted to a signed integer of the same size, the bit pattern does not change. However, the value it represents changes if the sign bit is set.

Unsigned integer values are converted to floating values by first converting to a signed integer of the same size, then converting that signed value to a floating value.

in Table 5.3 summarizes conversions from unsigned integral types.

Table 5.3

Conversions from Unsigned Integral Types

From	To	Method
unsigned char	**char**	Preserve bit pattern; high-order bit becomes sign bit
unsigned char	**short**	Zero-extend
unsigned char	**long**	Zero-extend
unsigned char	**unsigned short**	Zero-extend
unsigned char	**unsigned long**	Zero-extend
unsigned char	**float**	Convert to **long**; convert **long** to **float**

Table 5.3 *(continued)*

From	To	Method
unsigned char	**double**	Convert to **long**; convert **long** to **double**
unsigned short	**char**	Preserve low-order byte
unsigned short	**short**	Preserve bit pattern; high-order bit becomes sign bit
unsigned short	**long**	Zero-extend
unsigned short	**unsigned char**	Preserve low-order byte
unsigned short	**unsigned long**	Zero-extend
unsigned short	**float**	Convert to **long**; convert **long** to **float**
unsigned short	**double**	Convert to **long**; convert **long** to **double**
unsigned long	**char**	Preserve low-order byte
unsigned long	**short**	Preserve low-order word
unsigned long	**long**	Preserve bit pattern; high-order bit becomes sign bit
unsigned long	**unsigned char**	Preserve low-order byte
unsigned long	**unsigned short**	Preserve low-order word
unsigned long	**float**	Convert to **long**; convert **long** to **float**
unsigned long	**double**	Convert to **long**; convert **long** to **double**

Note

The **unsigned int** type is equivalent either to the **unsigned short** type or to the **unsigned long** type, depending on the implementation. Conversion of an **unsigned int** value proceeds the same as for an **unsigned short** or an **unsigned long**, whichever is appropriate.

5.7.1.3 Conversions from Floating-Point Types

A **float** value converted to a **double** value undergoes no change in value. A **double** value converted to a **float** value is represented exactly, if possible. Precision is lost if the value is too large to fit into a **float**, precision is lost.

A floating value is converted to an integer value by converting to a **long**. Conversions to other integer types occur the same as for a **long**. The decimal portion of the floating value is discarded in the conversion to a **long**; if the result is still too large to fit into a **long**, the result of the conversion is undefined.

Table 5.4 summarizes conversions from floating types.

Table 5.4

Conversions from Floating-Point Types

From	To	Method
float	**char**	Convert to **long**; convert **long** to **char**
float	**short**	Convert to **long**; convert **long** to **short**
float	**long**	Truncate at decimal point; if result is too large to be represented as **long**, result is undefined
float short	**unsigned short**	Convert to **long**; convert **long** to **unsigned**
float long	**unsigned long**	**Convert to long**; convert **long** to **unsigned**
float	**double**	**Change internal representation**
double	**char**	**Convert to float**; convert **float** to **char**
double	**short**	Convert to **float**; convert **float** to **short**
double	**long**	Truncate at decimal point; if result is too large to be represented as **long**, result is undefined
double short	**unsigned short**	Convert to **long**; convert **long** to **unsigned**
double long	**unsigned long**	**Convert to long**; convert **long** to **unsigned**
double	**float**	**Represent as a float**. If **double** value cannot be represented exactly as **float**, loss of precision occurs; if value is too large to be represented as **float**, the result is undefined

5.7.1.4 Conversions to and from Pointer Types

A pointer to one type of value can be converted to a pointer to a different type. However, the result may be undefined because of the alignment requirements and sizes of different types in storage.

A pointer to **void** may be converted to or from a pointer to any type, without restriction.

In some implementations, you can use the special keywords **near**, **far**, and **huge** to change the size of pointers within a program. The conversion path depends on your implementation. For example, on an 8086 processor, the compiler might use a segment-register value to convert a 16-bit pointer to a 32-bit pointer. See your User's Guide for information about pointer conversions.

A pointer value can also be converted to an integral value. The conversion path depends on size of the pointer and the size of the integral type, according to the following rules:

- If the size of the pointer is the greater than or equal to the size of the integral type, the pointer behaves like an unsigned value in the conversion, except that it cannot be converted to a floating value.
- If the pointer is smaller than the integral type, the pointer is first converted to a pointer with the same size as the integral type, then converted to the integral type. The implementation determines how a pointer is converted to a longer pointer; see your User's Guide for information about pointer conversions.

Conversely, an integral type can be converted to a pointer type according to the following rules:

- If the integral type is the same size as the pointer type, the conversion simply causes the integral value to be treated as a pointer (an unsigned integer).

If the size of the integral type is different from the size of the pointer type, the integral type is first converted to the size of the pointer, using the conversion paths given in Tables 5.2 and 5.3. It is then treated as a pointer value.

If the special keywords **near**, **far**, and **huge** are implemented, implicit conversions may be made on pointer values. In particular, the compiler may make assumptions about the default size of pointers and convert passed pointer values accordingly, unless a forward declaration is present to override the implicit conversion. See your User's Guide for information about pointer conversions.

5.7.1.5 Conversions from Other Types

Since an **enum** value is an **int** value by definition, conversions to and from an **enum** value are the same as those for the **int** type. An **int** is equivalent to either a **short** or a **long**, depending on the implementation.

No conversions between structure or union types are allowed.

The **void** type has no value, by definition. Therefore, it cannot be converted to any other type, and other types cannot be converted to **void** by assignment. However, you can explicitly cast a value to **void** type, as discussed in Section 5.7.2.

5.7.2 Type-Cast Conversions

You can use type casts to explicitly convert types. A type cast has the form

(*type-name*)*operand*

where *type-name* is a type and *operand* is a value to be converted to that type. (Type names are discussed in Section 4.9.)

The *operand* is converted as though it had been assigned to a variable of the *type-name* type. The conversion rules for assignments (outlined in Section 5.7.1) apply to type casts as well.

You can use the type name **void** in a cast operation, but you cannot assign the resulting expression to any item.

5.7.3 Operator Conversions

The conversions performed by C operators depend on the operator and on the type of the operand or operands. Many operators perform the usual arithmetic conversions, outlined in Section 5.3.1.

C permits some arithmetic with pointers. In pointer arithmetic, integer values are converted to express memory positions. (See the discussions of additive operators, Section 5.3.6, and subscript expressions, Section 5.2.5, for more information.)

5.7.4 Function-Call Conversions

The type of conversion performed on the arguments in a function call depends on the presence of a function prototype (forward declaration) with declared argument types for the called function.

If a function prototype is present, and it includes declared argument types, the compiler performs type checking. The type-checking process is outlined in detail in Chapter 7, "Functions."

If no function prototype is present, or if an old-style forward declaration omits the argument-type list, only the usual arithmetic conversions are performed on the arguments in the function call. These conversions are performed independently on each argument in the call. This means that a **float** value is converted to a **double**; a **char** or **short** value is converted to an **int**; and an **unsigned char** or **unsigned short** is converted to an **unsigned int**.

If the special keywords **near**, **far**, and **huge** are implemented, implicit conversions may also be made on pointer values passed to functions. You can override these implicit conversions by providing function prototypes to allow the compiler to perform type checking. See your User's Guide for information about pointer conversions.

Chapter 6
Statements

6.1 Introduction

The statements of a C program control the flow of program execution. In C, as in other programming languages, several kinds of statements are available to perform loops, to select other statements to be executed, and to transfer control. This chapter describes C statements in alphabetical order, as follows:

break statement

compound statement

continue statement

do statement

expression statement

for statement

goto statement

if statement

null statement

return statement

switch statement

while statement

C statements consist of keywords, expressions, and other statements. The following keywords appear in C statements:

break	**default**	**for**	**return**
case	**do**	**goto**	**switch**
continue	**else**	**if**	**while**

The expressions in C statements are the expressions discussed in Chapter 5 of this manual. Statements appearing within C statements may be any of the statements discussed in this chapter. A statement that forms a component of another statement is called the "body" of the enclosing statement. Frequently the statement body is a "compound" statement: a single statement composed of one or more statements.

The compound statement is delimited by braces ({ }); all other C statements end with a semicolon.

Any C statement may begin with an identifying label consisting of a name and a colon. Since only the **goto** statement recognizes statement labels, statement labels are described along with the **goto** statement in Section 6.8.

When a C program is executed, its statements are executed in the order in which they appear in the program, except where a statement explicitly transfers control to another location.

6.2 The break Statement

Syntax

break;

Execution

The **break** statement terminates the execution of the smallest enclosing **do**, **for**, **switch**, or **while** statement in which it appears. Control passes to the statement that follows the terminated statement. A **break** statement can appear only within a **do**, **for**, **switch**, or **while** statement.

Within nested statements, the **break** statement terminates only the **do**, **for**, **switch**, or **while** statement that immediately encloses it. You can use a **return** or **goto** statement to transfer control out of the nested structure.

Example

```
for (i = 0; i < LENGTH; i++) {
     for (j = 0; j < WIDTH; j++) {
          if (lines[i][j] == '\0') {
               lengths[i] = j;
               break;
          }
     }
}
```

This example processes an array of variable-length strings stored in `lines`. The **break** statement causes an exit from the interior **for** loop after the terminating null character (**\0**) of each string is found and its position is stored in `lengths [i]`. Control then returns to the outer **for** loop. The variable `i` is incremented and the process is repeated until `i` is greater than or equal to LENGTH.

6.3 The Compound Statement

Syntax

{
⟦*declaration*⟧
.
.
.
statement
⟦*statement*⟧
.
.
.
}

Execution

A compound statement typically appears as the body of another statement, such as the **if** statement. When a compound statement is executed, its statements are executed in the order in which they appear, except where a statement explicitly transfers control to another location. Chapter 4 of this manual describes the form and meaning of the declarations that can appear at the head of a compound statement.

Labeled Statements

Like other C statements, any of the statements in a compound statement can carry a label. Thus, you can use a **goto** statement to transfer into a compound statement. However, transferring into a compound statement is dangerous when the compound statement includes declarations that initialize variables. Since declarations appear before the executable statements in a compound statement, transferring directly to an executable statement within the compound statement bypasses the initializations. The results of such a transfer of control are undefined.

Example

```
if (i > 0) {
     line[i] = x;
     x++;
     i--;
}
```

In this example, if `i` is greater than 0, all of the statements in the compound statement are executed in order.

6.4 The continue Statement

Syntax

continue;

Execution

The **continue** statement passes control to the next iteration of the **do**, **for**, or **while** statement in which it appears, bypassing any remaining statements in the **do**, **for**, or **while** statement body. The next iteration of a **do**, **for**, or **while** statement is determined as follows:

- Within a **do** or a **while** statement, the next iteration starts by reevaluating the expression of the **do** or **while** statement.
- Within a **for** statement, the next iteration starts by evaluating the loop expression of the **for** statement. Then it evaluates the conditional expression and, based on the result, either terminates or iterates the statement body. (The **for** statement is discussed in Section 6.7.)

Example

```
while (i-- > 0) {
     x = f(i);
     if (x == 1)
          continue;
     y += x * x;
}
```

In this example, the statement body is executed if `i` is greater than 0. First `f(i)` is assigned to `x`; then, if `x` is equal to 1, the **continue** statement is executed. The rest of the statements in the body are ignored, and execution resumes at the top of the loop with the evaluation of `i-- > 0`.

6.5 The do Statement

Syntax

do
    statement
while (*expression*);

Execution

The body of a **do** statement is executed one or more times until *expression* becomes false (0). Execution proceeds as follows:

1. The statement body is executed.
2. The *expression* is evaluated. If *expression* is false, the **do** statement terminates and control passes to the next statement in the program. If *expression* is true (non-zero), the process is repeated, beginning with step 1.

The **do** statement may also terminate when a **break**, **goto**, or **return** statement is executed within the statement body.

Example

```
do {
     y = f(x);
     x--;
} while (x > 0);
```

In this **do** statement, the two statements `y = f(x);` and `x--;` are executed, regardless of the initial value of x. Then `x > 0` is evaluated. If x is greater than 0, the statement body is executed again and `x > 0` is reevaluated. The statement body is executed repeatedly as long as x remains greater than 0. Execution of the **do** statement terminates when x becomes 0 or negative. The body of the loop is executed at least once.

6.6 The Expression Statement

Syntax

expression;

Execution

When an expression statement is executed, the expression is evaluated according to the rules outlined in Chapter 5 of this manual.

In C, assignments are expressions. The value of the expression is the value being assigned (sometimes called the "right-hand value").

Function calls are also considered expressions. The value of the expression is the value, if any, returned by the function. If a function returns a value, the expression statement usually includes an assignment to store the returned value when the function is called. The value returned by the function is usually used as an operand in another expression. If the value is to be used more than once, it can be assigned to another variable. If the value is neither used as an operand nor assigned, the function is called but the return value, if any, is not used.

Examples

```
/******************** Example 1 ********************/
x = (y + 3);
```

In example 1, x is assigned the value of y + 3.

```
/******************** Example 2 ********************/
x++;
```

In example 2, x is incremented.

```
/******************** Example 3 ********************/
z = f(x) + 3;
```

Example 3 shows a function-call expression. The value of the expression, which includes any value returned by the function, is assigned to the variable z.

6.7 The for Statement

Syntax

for (⟦*init-expression* ⟧; ⟦ *cond-expression* ⟧; ⟦*loop-expression*⟧)
    statement

Execution

The body of a **for** statement is executed zero or more times until the optional *cond-expression* becomes false. You can use the optional *init-expression* and *loop-expression* to initialize and change values during the **for** statement's execution.

Execution of a **for** statement proceeds as follows:

1. The *init-expression,* if any, is evaluated.
2. The *cond-expression,* if any, is evaluated. Three results are possible:
    a. If the *cond-expression* is true (nonzero), the *statement* is executed; then the *loop-expression,* if any, is evaluated. The process then begins again with the evaluation of *cond-expression.*
    b. If the *cond-expression* is omitted, the *cond-expression* is considered true, and execution proceeds exactly as described for case a. A **for** statement without a *cond-expression* terminates only when a **break** or **return** statement within the statement body is executed, or when a **goto** to a labeled statement outside the **for** statement body is executed.
    c. If the *cond-expression* is false, execution of the **for** statement terminates and control passes to the next statement in the program.

A **for** statement also terminates when a **break, goto,** or **return** statement within the statement body is executed.

Example

```
for (i = space = tab = 0; i < MAX; i++) {
     if (line[i] == ' ')
          space++;
     if (line[i] == '\t') {
          tab++;
          line[i] = ' ';
     }
}
```

This example counts space ('\x20') and tab ('\t') characters in the array of characters named line and replaces each tab character with a space. First i, space, and tab are initialized to 0. Then i is compared to the constant MAX; if i is less than MAX, the statement body is executed. Depending on the value of line[i], the body of one or neither of the **if** statements is executed. Then i is incremented and tested against MAX; the statement body is executed repeatedly as long as i is less than MAX.

6.8 The goto and Labeled Statements

Syntax

goto *name*;
.
.
.
name: *statement*

Execution

The **goto** statement transfers control directly to the statement that has *name* as its label. The labeled statement is executed immediately after the **goto** statement is executed. A statement with the given label must reside in the same function, and the given label can appear before only one statement in the same function.

A statement label is meaningful only to a **goto** statement; in any other context, a labeled statement is executed without regard to the label.

Forming Labels

A label name is simply an identifier. (Section 2.4 describes the rules that govern the construction of identifiers.) Each statement label must be distinct from other statement labels in the same function.

Labeled Statements

Like other C statements, any of the statements in a compound statement can carry a label. Thus, you can use a **goto** statement to transfer into a compound statement. However, transferring into a compound statement is dangerous when the compound statement includes declarations that initialize variables. Since declarations appear before the executable statements in a compound statement, transferring directly to an executable statement within the compound statement bypasses the initializations. The results are unpredictable.

Example

```
if (errorcode > 0)
     goto exit;
.
.
.
exit:
     return (errorcode);
```

In this example, a **goto** statement transfers control to the point labeled `exit` if an error occurs.

6.9 The if Statement

Syntax

if (*expression***)**
   statement1
⟦ **else**
   statement2 ⟧

Execution

The body of an **if** statement is executed selectively, depending on the value of *expression*:

1. The *expression* is evaluated.
   a. If *expression* is true (nonzero), *statement1* is executed.
   b. If *expression* is false, *statement2* is executed.
   c. If *expression* is false and the **else** clause is omitted, *statement1* is ignored.
2. Control passes from the **if** statement to the next statement in the program.

Example

```
if (i > 0)
     y = x/i;
else {
     x = i;
     y = f(x);
}
```

In this example, the statement `y = x/i;` is executed if `i` is greater than 0. If `i` is less than or equal to 0, `i` is assigned to `x` and `f(x)` is assigned to `y`. Note that the statement forming the **if** clause ends with a semicolon.

Nesting

C does not offer an "else if" statement, but you can achieve the same effect by nesting **if** statements. An **if** statement may be nested within either the **if** clause or the **else** clause of another **if** statement.

When nesting **if** statements and **else** clauses, use braces to group the statements and clauses into compound statements that clarify your intent. If no braces are present, the compiler resolves ambiguities by pairing each **else** with the most recent **if** lacking an **else**.

Examples

```
/******************** Example 1 ********************/

if (i > 0)              /* Without braces */
     if (j > i)
          x = j;
     else
          x = i;
```

In example 1, the **else** clause is associated with the inner **if** statement. If `i` is less than or equal to 0, no value is assigned to `x`.

```
/******************** Example 2 ********************/

if (i > 0) {            /* With braces */
     if (j > i)
          x = j;
}
```

```
else
     x = i;
```

In example 2, the braces surrounding the inner **if** statement make the **else** clause part of the outer **if** statement. If `i` is less than or equal to 0, `i` is assigned to `x`.

6.10 The Null Statement

Syntax

;

Execution

A null statement is a statement containing only a semicolon; it may appear wherever a statement is expected. Nothing happens when a null statement is executed.

Statements such as **do, for, if,** and **while** require that an executable statement appear as the statement body. The null statement satisfies the syntax requirement in cases that do not need a substantive statement body.

Example

```
for (i = 0; i < 10; line[i++] = 0)
    ;
```

In this example, the loop expression of the **for** statement (`line[i++]=0`) initializes the first 10 elements of `line` to 0. The statement body is a null statement, since no further statements are necessary.

Labeling a Null Statement

As with any other C statement, you can include a label before a null statement. To label an item that is not a statement, such as the closing brace of a compound statement, you can label a null statement and insert it immediately before the item to get the same effect.

6.11 The return Statement

Syntax

return ⟦*expression*⟧;

Execution

The **return** statement terminates the execution of the function in which it appears and returns control to the calling function. Execution resumes in the calling function at the point immediately following the call. The value of *expression*, if present, is returned to the calling function. If *expression* is omitted, the return value of the function is undefined.

By convention, parentheses enclose the *expression* of the **return** statement. However, C does not require the parentheses.

Example

```
main()
{
     void draw(int,int);
     long sq(int);
     .
     .
     .
     y = sq(x);
     draw(x, y);
     .
     .
     .
}

long sq(x)
int x;
{
     return (x * x);
}

void draw(x,y)
int x, y;
{
     .
     .
```

```
        .
        return;
}
```

In this example, the main function calls two functions: sq and draw. The sq function returns the value of x * x to main, where the return value is assigned to y. The draw function is declared as a **void** function and does not return a value. An attempt to assign the return value of draw would cause a diagnostic message to be issued.

Omitting the Return Statement

If no **return** statement appears in a function definition, control automatically returns to the calling function after the last statement of the called function is executed. The return value of the called function is undefined. If a return value is not required, declare the function to have **void** return type.

6.12 The switch Statement

Syntax

```
switch (expression) {
    [[declaration]]
    .
    .
    .
    [[case constant-expression :]]
    .
    .
    .
        [[statement]]
        .
        .
        .
    [[default :
        statement]]
}
```

Execution

The **switch** statement transfers control to a statement within its body. Control passes to the statement whose **case** *constant-expression* matches the value of the **switch** *expression*. Execution of the statement body begins at the selected statement and proceeds until the end of the body or until a statement transfers control out of the body.

The **default** statement is executed if no **case** *constant-expression* is equal to the value of the **switch** *expression*. If the **default** statement is omitted, and no **case** match is found, none of the statements in the **switch** body is executed. The **default** statement need not come at the end, it can appear anywhere in the body of the **switch** statement.

The type of the **switch** *expression* must be integral, but the resulting value is converted to an **int**. Each **case** *constant-expression* is then converted using the usual arithmetic conversions. The value of each **case** *constant-expression* must be unique within the statement body. If the type of the **switch** *expression* is larger than **int**, a diagnostic message is issued.

The **case** and **default** labels of the **switch** statement body are significant only in the initial test that determines where execution starts in the statement body. All statements between the statement where execution starts and the end of the body are executed regardless of their labels, unless a statement transfers control out of the body entirely.

Note

Declarations may appear at the head of the compound statement forming the **switch** body, but initializations included in the declarations are not performed. The **switch** statement transfers control directly to an executable statement within the body, bypassing the lines that contain initializations.

Examples

```
/******************** Example 1 ********************/

switch (c) {
        case 'A':
             capa++;
        case 'a':
             lettera++;
        default :
             total++;
      }
```

In example 1, all three statements of the **switch** body are executed if `c` is equal to `'A'`. Execution control is transferred to the first statement (`capa++;`) and continues in order through the rest of the body. If `c` is equal to `'a'`, `lettera` and `total` are incremented. Only `total` is incremented if `c` is not equal to `'A'` or `'a'`.

```
/******************** Example 2 ********************/

switch (i) {
        case -1:
            n++;
            break;
        case 0 :
            z++;
            break;
        case 1 :
            p++;
            break;
    }
```

In example 2, a **break** statement follows each statement of the **switch** body. The **break** statement forces an exit from the statement body after one statement is executed. If `i` is equal to −1, only `n` is incremented. The **break** following the statement `n++;` causes execution control to pass out of the statement body, bypassing the remaining statements. Similarly, if `i` is equal to 0, only `z` is incremented; if `i` is equal to 1, only `p` is incremented. The final **break** statement is not strictly necessary, since control passes out of the body at the end of the compound statement, but it is included for consistency.

Multiple Labels

A statement may carry multiple **case** labels, as the following example shows:

```
case 'a' :
case 'b' :
case 'c' :
case 'd' :
case 'e' :
case 'f' :  hexcvt(c);
```

Although you can label any statement within the body of the **switch** statement, no statement is required to carry a label. You can freely intermingle statements with and without labels. Keep in mind, however, that once the **switch** statement passes control to a statement within the body, all following statements in the block are executed, regardless of their labels.

6.13 The while Statement

Syntax

while (*expression*)
    statement

Execution

The body of a **while** statement is executed zero or more times until *expression* becomes false (0). Execution proceeds as follows:

1. The *expression* is evaluated.
2. If the
   expression is initially false, the body of the **while** statement is never
   executed, and control passes from the **while** statement to the next statement in the program.

   If *expression* is true (nonzero), the body of the statement is executed and the process is repeated beginning at step 1.

The **while** statement may also terminate when a **break**, **goto**, or **return** within the statement body is executed.

Example

```
while (i >= 0) {
     string1[i] = string2[i];
     i--;
}
```

This example copies characters from `string2` to `string1`. If `i` is greater than or equal to 0, `string2[i]` is assigned to `string1[i]` and `i` is decremented. When `i` reaches or falls below 0, execution of the **while** statement terminates.

Chapter 7
Functions

7.1 Introduction

A function is an independent collection of declarations and statements, usually designed to perform a specific task. C programs have at least one function, the **main** function, and they may have other functions. This chapter describes how to define, declare, and call C functions.

A function *definition* specifies the name of the function, the types and number of its formal parameters, and the declarations and statements that determine what it does. These declarations and statements are called the "function body." The function definition also gives the function's return type and its storage class. If the return type and storage class are not stated explicitly, they default to **int** and **extern**, respectively.

A function *prototype* (or declaration) establishes the name, return type, and storage class of a function fully defined elsewhere in the program. It can also include declarations giving the types and number of the function's formal parameters, and can even name the formal parameters, although such names go out of scope at the end of the declaration. The storage class **register** can also specified for a formal parameter.

The function prototype has the same form as a function definition, except that the prototype ends with a semicolon instead of a function body. The following example contrasts the current prototype formats with the old forms of function declaration and definition:

Example

```
double *function(int a, double *x) ;     /* Function
                                          * Prototype
                                         */
double *unction (int, double *) ;        /* Old Form of
        .                                 * Declaration
        .                                */
        .
double *function (int a, double *real) ; /* Prototype-style
        {                                 * Function
          return (*real + a) ;            * Definition
        }                                */

double *unction (x , y)                  /* Old Form of a
        double *y ;                       * Function
        int x ;                           * Definition
```

```
    {                                              */
        return (*y + a) ;
    }
```

The example illustrates why the concise and clear prototype formats are preferred to the old forms. Note also the close resemblance between the prototype and prototype-style definition.

The compiler uses the prototype or declaration to compare the types of actual arguments in subsequent calls to the function with the function's formal parameters, even in the absence of an explicit definition of the function. Explicit prototypes and declarations are optional for functions whose return type is **int**. However, to ensure correct behavior, you must declare or define functions with other return types before calling them. (Function prototype declarations are discussed further in Section 7.3 and in Chapter 4, "Declarations.")

If no prototype or declaration is provided, a default prototype is created from information provided in the first occurrence of the function name, whether that is a call or definition. However, such a default prototype may not adequately represent a subsequent definition of, or call to, the function.

A function *call* passes execution control from the calling function to the called function. The actual arguments, if any, are passed by value to the called function. Execution of a **return** statement in the called function returns control and possibly a value to the calling function.

7.2 Function Definitions

Syntax

⟦*sc-specifier*⟧⟦*type-specifier*⟧ *declarator* **(**⟦*formal-parameter-list*⟧**)**
function-body

A function definition specifies the name, formal parameters, and body of a function. It can also stipulate the function's return type and storage class.

The optional *sc-specifier* gives the function's storage class, which must be either **static** or **extern**.

The optional *type-specifier* and mandatory *declarator* together specify the function's return type and name. The *declarator* is a combination of the identifier that names the function and the parentheses following the function name.

Formal parameter declarations are included in the optional *formal-parameter-list* in the parentheses following the function name. The following syntax illustrates the form of each parameter in the *formal-parameter-list.*

⟦⟦**register**⟧ *type-specifier* ⟦*declarator*⟦⟦, ...⟧⟦,...⟧⟧⟧

A *formal-parameter-list* contains declarations for the function's parameters. If no arguments will be passed to the function, the list contains the keyword **void**. Otherwise, it may contain a full or partial list of formal parameters. If the list is partial, it is terminated by the "ellipsis notation," a comma followed by three periods (**,...**).The ellipsis notation indicates there may be more arguments passed to the function. Without the ellipsis notation, the behavior of a function is undefined if it receives parameters in addition to those declared in the *formal-parameter-list.* When a prototype is available, argument checking and conversion are automatically performed. If no information is given concerning the formal parameters, any undeclared arguments simply undergo the usual arithmetic conversions.

The *type-specifier* can be omitted only if **register** storage class is specified for a value of **int** type.

The *function-body* is a compound statement containing local variable declarations, references to externally declared items, and statements.

Note

The old form for function declaration and definition is still supported, but is considered obsolescent. Use of the prototype form is recommended in new code. The old function-definition form is represented in the following syntax:

⟦ *sc-specifier* ⟧⟦ *type-specifier* ⟧ *declarator* **(** ⟦ *identifier-list* ⟧ **)**
⟦*parameter-declarations*⟧
function-body

The *identifier-list* is an optional list of identifiers that the function will use as the names of formal parameters. The *parameter-declarations* establish the types of the formal parameters.

Sections 7.2.1–7.2.4 describe the parts of a function definition in detail.

7.2.1 Storage Class

The storage-class specifier in a function definition gives the function either **extern** or **static** storage class. If a function definition does not include a storage-class specifier, the storage class defaults to **extern**. You can explictly give the **extern** storage-class specifier in a function definition, but it is not required.

A function with **static** storage class is visible only in the source file in which it is defined. All other functions, whether they are given **extern** storage class explicitly or implicitly, are visible throughout all the source files that make up the program.

The storage-class specifier is required in a function definition in only one case: if the function is declared elsewhere in the same source file with the **static** storage-class specifier.

If **static** storage class is desired, it must be declared on first occurrence of a declaration or definition of the function.

Note

A Microsoft extension to the ANSI C standard offers some lattitude on functions declared without a storage-class specifier. When the extensions are enabled, a function originally declared without a storage class will be given **static** storage class if the function definition is in the same source file, and explicitly specifies **static** storage class. For information on enabling and disabling extensions, see your User's Guide.

7.2.2 Return Type and Function Name

The return type of a function defines the size and type of the value returned by the function. The type declaration has the form

⟦ *type-specifier* ⟧ *declarator*

where the *type-specifier*, together with the *declarator*, defines the function's return type and name.

The *type-specifier* can specify any fundamental, structure, or union type. If you do not include a *type-specifier*, the return type **int** is assumed.

The *declarator* is the function identifier, which may be modified to a pointer type. The parentheses following the identifier establish the item as a function. Functions cannot return arrays or functions, but they can return pointers to any type, including arrays and functions.

The return type given in the function definition must match the return type in declarations of the function elsewhere in the program. You need not declare functions with **int** return type before you call them. However, functions with other return types must be defined or declared before they are called.

A function's return type is used only when the function returns a value, which occurs when a **return** statement containing an expression is executed. The expression is evaluated, converted to the return value type if necessary, and returned to the point at which the function was called. If no **return** statement is executed, or if the **return** statement does not contain an expression, the return value is undefined. If the calling function expects a return value, the behavior of the program is also undefined.

Examples

```
/******************** Example 1 ********************/

     /* prototype-style definition: */
    static add (register x, int y)
     {
        return (x+y);
     }
     /* old-style definition: */
    subtract (x , y)
     {
        return (x-y);
    }
```

In Example 1, the return type of add is **int** by default. The function has **static** storage class, which means that only functions in the same source file can call it. The formal parameters declared in the header include one **int**, x, for which register storage is requested, and a second **int**, y. The second function, subtract is defined in the old form. Its return type is **int** by default, and because it has no formal parameter declarations, the identifiers x and y are assumed to have **int** type by default.

```
/********************** Example 2 *********************/

    typedef struct  {
        char name[20];
        int id;
        long class;
    } STUDENT;

          /* return type is STUDENT: */
    STUDENT sortstu (STUDENT a, STUDENT b)
    {
        return ( (a.id < b.id) ? a : b );
    }
```

The second example defines the STUDENT type with a **typedef** declaration and defines the function sortstu to have STUDENT return type. The function selects and returns one of its two structure arguments. This prototype-style definition has the formal parameters declared in the header. In subsequent calls to the function the compiler will check to make sure the argument types are STUDENT. Efficiency could be enhanced by passing pointers to the structure elements, rather than the values themselves.

```
/********************** Example 3 *********************/

      /* return type is char pointer: */
    char *smallstr(s1, s2)
    char s1[], s2[];
    {
        int i;

        i=0;
        while ( s1[i] != '\0' && s2[i] != '\0' )
```

```
                    i++;
        if ( s1[i] == '\0' )
                return (s1);
        else
                return (s2);
    }
```

Example 3 uses the old form to define a function returning a pointer to an array of characters. The function takes two character arrays (strings) as arguments and returns a pointer to the shorter of the two strings. A pointer to an array points to the type of the array elements; thus, the return type of the function is pointer to **char**.

7.2.3 Formal Parameters

Formal parameters are variables that receive values passed to a function by a function call. In a function prototype-style definition, the parentheses following the function name contain complete declarations of the function's formal parameters.

Note

In the old form of a function definition, the formal parameters were declared following the closing parenthesis, immediately before the beginning of the compound statement constituting the function body. In that form, an *identifier-list* within the parentheses specifies the name of each of the formal parameters and the order in which they take on values in the function call. The *identifier-list* consists of zero or more identifiers, separated by commas. The list must be enclosed in parentheses, even if it is empty. This form is obsolescent and should not be used in new code.

If at least one formal parameter occurs in the *formal-parameter-list,* the list can end with a comma followed by three periods (**,...**). This construction, called the "ellipsis notation," indicates a variable number of arguments to the function. However, a call to the function is expected to have at least as many arguments as there are formal parameters before the last comma. In the obsolescent definition form, the ellipsis notation can follow the last identifier in the *identifier-list.*

If no arguments will be passed to the function, the list of formal parameters is replaced by the keyword **void**. This use of **void** is distinct from its use as a type specifier.

Note

To maintain compatibility with previous versions, a Microsoft extension to the ANSI C standard allows a comma without trailing periods (,) at the end of the list of formal parameters to indicate a variable number of arguments. See your User's Guide for information on enabling and disabling extensions.

Formal parameter declarations specify the types, sizes, and identifiers of values stored in the formal parameters. In the obsolescent function definition form, these declarations have the same form as other variable declarations (see Section Chapter 4, "Declarations"). However, in a function prototype-style definintion, each identifier in the *formal-parameter-list* must be preceded by its appropriate type specifier. For example, in the following (obsolescent form) definition of the function `old`, `double x, y, z ;` can be declared simply by separating identifiers with commas:

```
void old(x, y, z)
    double x, y, z ;
        {
          ;
        }
void new(double x, double y, double z)
        {
          ;
        }
```

The function called new is defined in prototype format, with a list of formal parameters in the parentheses. In this form, the type specifier `double` has to be repeated for each identifier.

The order and type of formal parameters, including any use of the ellipsis notation, must be the same in the all function declarations (if any) as in the function definition. The order and type must also be the same in arguments specified in calls to the function, up the the point of the ellipsis notation. Arguments following the ellipsis are not checked. A formal parameter can have any fundamental, structure, union, pointer, or array type.

The only storage class you can specify for a formal parameter is **register**. Undeclared identifiers in the parentheses following the function name are assumed to have **int** type. In the old function-definition form, formal parameter declarations can be in any order.

The identifiers of the formal parameters are used in the function body to refer to the values passed to the function. These identifiers cannot be redefined in the outermost block of the function body, but they may be redefined in inner, nested blocks.

In the obsolescent form, only identifiers appearing in the identifier list can be declared as formal parameters. Functions having variable length argument lists should use the new prototype form. You are responsible for determining the number of arguments passed and for retrieving additional arguments from the stack within the body of the function. (See your User's Guide for information about macros that allow you to do this in a portable way.)

The compiler performs the usual arithmetic conversions independently on each formal parameter and on each actual argument, if necessary. After conversion, no formal parameter is shorter than an **int**, and no formal parameter has **float** type. This means, for example, that declaring a formal parameter as a **char** has the same effect as declaring it as an **int**.

If the **near**, **far**, and **huge** keywords are implemented, the compiler may also convert pointer arguments to the function. The conversions performed depend on the default size of pointers in the program and the presence or absence of a list of argument types for the function. See your User's Guide for specific information about pointer conversions.

The converted type of each formal parameter determines the interpretation of the arguments that the function call places on the stack. A type mismatch between an actual argument and a formal parameter may cause the arguments on the stack to be misinterpreted. For example, if a 16-bit pointer is passed as an actual argument, then declared as a **long** formal parameter, the first 32 bits on the stack are interpreted as a **long** formal parameter. This error creates problems not only with the **long** formal parameter, but with any formal parameters that follow it. You can detect errors of this kind by declaring function prototypes for all functions.

Example

```
struct student {
        char name[20];
        int id;
        long class;
        struct student *nextstu;
} student;

main()
{
        /* declaration of function prototype: */
        int match ( struct student *r, char *n );
      .
      .
      .
        if (match (student.nextstu, student.name) > 0) {
      .
      .
      .
        }
}

/* prototype style function definition */
match ( struct student *r, char *n )
{
        int i = 0;

        while ( r->name[i] == n[i] )
                if ( r->name[i++] == '\0' )
                        return (r->id);
        return (0);
}
```

The example contains a structure-type declaration, a forward declaration of the function match, a call to match, and the definition of match. Note that the same name, student, can be used without conflict both for the structure tag and for the structure variable name.

The match function is declared to have two arguments: the first, represented by r, is a pointer to the struct student type; the second, represented by n, is a pointer to a value of type **char**.

The two formal parameters of the match function are declared in the formal parameter list in the parentheses following the function name, with the identifiers r and n. The parameter r is declared as a pointer to the struct student type; the parameter n is declared as a pointer to a **char** type.

The function is called with two arguments, both members of the `student` structure. Because there is a forward declaration of `match`, the compiler performs type checking between the actual arguments and the the types specified in the prototype declaration and between the actual arguments and the formal parameters. Since the types match, no warnings or conversions are necessary.

Note that the array name given as the second argument in the call evaluates to a **char** pointer. The corresponding formal parameter is also declared as a **char** pointer and is used in subscripted expressions as though it were an array identifier. Since an array identifier evaluates to a pointer expression, the effect of declaring the formal parameter as `char *n` is the same as declaring it `char n[]`.

Within the function, the local variable `i` is defined and used to monitor the current position in the array. The function returns the `id` structure member if the `name` member matches the array `n`; otherwise, it returns 0.

7.2.4 Function Body

A function body is a compound statement containing the statements that define what the function does. It may also contain declarations of variables used by these statements. (See Section 6.3 for a discussion of compound statements.)

All variables declared in a function body have **auto** storage class unless otherwise specified. When the function is called, storage is created for the local variables and local initializations are performed. Execution control passes to the first statement in the compound statement and continues sequentially until a **return** statement is executed or the end of the function body is encountered. Control then returns to the point at which the function was called.

A **return** statement containing an expression must be executed if the function is to return a value. The return value of a function is undefined if no **return** statement is executed or if the **return** statement does not include an expression.

7.3 Function Prototypes (Declarations)

A function prototype declaration specifies the name, return type, and storage class of a function. It may also establish types and identifiers of some or all of the function's arguments. The prototype has the same format as the function definition, except that it is terminated by a semicolon immediately following the closing parenthesis and therefore has no body. (See Chapter 4, "Declarations," for a detailed description of the syntax of function declarations.)

You can declare a function implicitly, or you can use a "function prototype" (forward declaration) to declare it explicitly. A prototype is a declaration that precedes the function definition. In either case, the return type must agree with the return type specified in the function definition.

You implicitly declare a function if a call to the function precedes its declaration or definition in the source file. In this case the C compiler constructs a default prototype of the function, giving it **int** return type, and using the types and number of the actual arguments as the basis for declaring the formal parameters.

A prototype declaration establishes the attributes of a function so that calls to it that precede its definition or occur in other source files can be checked for argument and return type mismatches. If you specify the **static** storage-class specifier in a forward declaration, you must also specify the **static** storage class in the function definition.

If you specify the **extern** storage-class specifier or omit the storage-class specifier entirely, the function has **extern** class. (See the *Note* in Section 7.2.1 for an explanation of the Microsoft extension that offers some lattitude in function storage-class specification.)

Function prototypes have the following important uses:

- They establish the return type for functions that return any type other than **int**. If you call such a function before you declare or define it, the results are undefined. Functions that return **int** values can also have prototype declarations, but do not require them.
- If the prototype contain a full list of parameter types, the types of the arguments occurring in a function call can be established. The prototype can include both the type of, and an identifier for, each

expression that will be passed as an actual argument. However, such identifiers have scope only until the end of the declaration. The prototype can also reflect the fact that the number of arguments will be variable.

The parameter list in a prototype is a list of type names, separated by commas, corresponding to the actual arguments in the function call. The list is used for checking the correspondence of actual arguments in the function call with the formal parameters in the function definition. Without such a list, mismatches may not be revealed, so the compiler cannot generate diagnostic messages concerning them. (Type checking is discussed further in Section 7.4.1, "Actual Arguments.")

- Forward declarations are used to initialize pointers to functions before those functions are defined.

Example

```
main()
{
        int a = 0, b = 1;
        float val1= 2.0, val2 = 3.0;

        /* function prototype: */
        double realadd(double x, double y);

        a = intadd (a, b);         /* first call to intadd */
        val1 = realadd(val1, val2);
        a = intadd(val1,b);        /* second call to intadd */
}

/* function defined with formal parameters in header: /*
intadd(int a, int b)
{
        return (a + b);
}

double realadd(double x, double y)
{
        return (x + y);
}
```

In this example, the function `intadd` is implicitly declared to return an **int** value, since it is called before it is defined. The compiler creates a prototype using the information in the first call. Therefore, when the second call to `intadd` is encountered, the compiler sees the mismatch between `val1`, which is a **float**, and the **int** type of the first argument in its self-created prototype. The **float** is converted to an **int** and passed. Note that if the calls to `intadd` were reversed, the prototype created would expect a **float** as the first argument to `intadd`. When the second call is made, the variable `a` would be converted at the call, but when the value is actually passed to `intadd`, a diagnostic would be issued because the **int** type specified in the definition does not match the **float** type in the compiler-created prototype.

The function `realadd` returns a **double** value instead of an **int**. Therefore, the prototype of `realadd` in the `main` function is necessary, because the `realadd` function is called before it is defined. Note that the definition of `realadd` matches the forward declaration by specifying the **double** return type.

The forward declaration of `realadd` also establishes the types of its two arguments. The actual argument types match the types given in the declaration and also match the types of the formal parameters.

7.4 Function Calls

Syntax

expression(⟦*expression-list*⟧)

A function call is an expression that passes control and actual arguments (if any) to a function. In a function call, *expression* evaluates to a function address and *expression-list* is a list of expressions (separated by commas). The values of these latter expressions are the actual arguments passed to the function. If the function takes no arguments, the *expression-list* can be empty.

When the function call is executed:

1. The expressions in *expression-list* are evaluated and converted using the usual arithmetic conversions. If a function prototype is available, the results of these conversions may be further converted consistent with the formal parameter declarations.
2. The expressions in *expression-list* are passed to the formal parameters of the called function. The first expression in the list always corresponds to the first formal parameter of the function, the second expression corresponds to the second formal parameter, and

so on through the list. Since the called function uses copies of the actual arguments, any changes it makes to the arguments do not affect the values of variables from which the copies may have been made.

3. Execution control passes to the first statement in the function.
4. The execution of a **return** statement in the body of the function returns control and possibly a value to the calling function. If no **return** statement is executed, control returns to the caller after the last statement of the called function is executed, and the return value is undefined.

Important

The expressions in the function argument list can be evaluated in any order, so arguments whose values may be affected by side effects from another argument have undefined values. The sequence point defined by the function-call operator guarantees only that all side effects in the argument list are evaluated before control passes to the called function. See Chapter 5, "Expressions and Assignments," for more information on sequence points.

The only requirement in a function call is that the expression before the parentheses must evaluate to a function address. This means that a function can be called through any function-pointer expression.

A function is called in much the same way it is declared. For instance, when you declare a function, you specify the name of the function, followed by a list of formal parameters in parentheses. Similarly, when a function is called, you need only specify the name of the function, followed by an argument list in parentheses. The indirection operator (*) is not required to call the function because the name of the function evaluates to the function address.

The same principle applies when you call a function using a pointer. For example, suppose a function pointer is declared as follows:

```
int (*fpointer)(int num1, int num2);
```

The identifier `fpointer` is declared to point to a function taking two **int** arguments, represented by `num1` and `num2`, respectively, and returning an **int** value. A function call using fpointer might look like this:

```
(*fpointer)(3,4)
```

The indirection operator (*) is used to obtain the address of the function to which fpointer points. The function address is then used to call the function. If a forward declaration of the pointer to the function precedes the call, the same checking will be performed as with any other function declaration.

Examples

```
/********************** Example 1 *********************/

    double *realcomp(double value1, double value2);
    double a, b, *rp;
    .
    .
    .
    rp = realcomp(a, b);
```

In Example 1, the realcomp function is called in the statement rp = realcomp(a, b);. Two **double** arguments are passed to the function. The return value, a pointer to a **double** value, is assigned to rp.

```
/********************* Example 2 *********************/

    main ()
    {
        /* non-prototype function declarations: */
        long lift(int), step(int), drop(int);

        /* prototype form of function declaration: */
        void work (int number, long (*function)(int i));
        int select, count;
        int i;
      .
      .
      .
        select = 1;
        switch ( select ) {
                case 1: work(count, lift);
                        break;

                case 2: work(count, step);
```

```
                        break;

                case 3: work(count, drop);

                default:
                        break;
        }
    }

    /* function definition with formal parameters in header: */
    void work ( int number, long (*function)(int i) )
    {
        int i;
        long j;

        for (i = j = 0; i < number; i++)
                j += (*function)(i);
    }
```

In Example 2, the function call

```
work (count, lift);
```

in `main` passes an integer variable and the address of the function `lift` to the function `work`. Note that the function address is passed simply by giving the function identifier, since a function identifier evaluates to a pointer expression. To use a function identifier in this way, the function must be declared or defined before the identifier is used; otherwise, the identifier is not recognized. In this case, a prototype declaration for `work` is given at the beginning of the `main` function.

The formal parameter `function` in `work` is declared to be a pointer to a function taking one **int** argument and returning a **long** value. The parentheses around the parameter name are required; without them, the declaration would specify a function returning a pointer to a **long** value.

The function `work` calls the selected function by using the following function call:

```
(*function)(i);
```

One argument, `i`, is passed to the called function.

7.4.1 Actual Arguments

An actual argument can be any value with fundamental, structure, union, or pointer type. Although you cannot pass arrays or functions as parameters, you can pass pointers to these items.

All actual arguments are passed by value. A copy of the actual argument is assigned to the corresponding formal parameter. The function uses this copy without affecting the variable from which it was originally derived.

Pointers provide a way for a function to access a value by reference. Since a pointer to a variable holds the address of the variable, the function can use this address to access the value of the variable. Pointer arguments allow a function to access arrays and functions, even though arrays and functions cannot be passed as arguments.

The expressions in a function call are evaluated and converted as follows:

- For each actual argument in the function call, the usual arithmetic conversions are performed on the argument. If a prototype is available, the resulting argument type is compared to the corresponding formal parameter. If they do not match, there is either a conversion performed, or a diagnostic message is issued. The formal parameters also undergo the usual arithmetic conversions.
- If no prototype is available, the results of the usual arithmetic conversions on the actual arguments are passed, and a prototype is created with formal parameter types corresponding to the results of the conversion.

If the **near**, **far**, and **huge** keywords are implemented, implementation-dependent conversions on pointer arguments may also be performed. See your User's Guide for information about pointer conversions.

The number of expressions in the expression list must match the number of formal parameters, unless the function's prototype declaration or definition explicitly specifies a variable number of arguments. In this case, the compiler checks as many arguments as there are type names in the list of formal parameters and converts them, if necessary, as described above.

If the declaration's formal parameter list or list of argument types contains only the keyword **void**, the compiler expects zero actual arguments in the function call and zero formal parameters. It produces a diagnostic message if it finds otherwise.

The type of each formal parameter also undergoes the usual arithmetic conversions. The converted type of each formal parameter determines how the arguments on the stack are interpreted; if the type of the formal parameter does not match the type of the actual argument, the data on the stack may be misinterpreted.

Note

Type mismatches between actual arguments and formal parameters can produce serious errors, particularly when the sizes are different. The compiler may not be able to detect these errors unless you declare complete prototypes of functions prior to calling them. In the absence of explicit prototypes, the compiler constructs prototypes from whatever information is available in the first reference to the function.

As an example of a serious error, consider a call to a function with an **int** argument. If the function is defined to take a **long**, and the definition occurs in a different module, the compiler-generated prototype will not match the definition, but there will be no detection of the error, because the separate modules will compile without diagnostic messages.

Example

```
main ()
{
        /* function prototype: */
        void swap (int *num1, int *num2);
        int x, y;
     .
     .
     .
        swap(&x, &y);
}

/* function definition: */
void swap (int *num1, int *num2)
{
        int t;

        t = *num1;
        *num1 = *num2;
        *num2 = t;
}
```

In this example, the swap function is declared in main to have two arguments, represented respectively by identifiers num1 and num2, both of which are pointers to integers. The formal parameters num1 and num2 in the prototype definition are also declared as pointers to integer variables. In the function call

```
swap (&x, &y)
```

the address of x is stored in num1 and the address of y is stored in num2. Now two names, or "aliases," exist for the same location. References to *num1 and *num2 in swap are effectively references to x and y in main. The assignments within swap actually exchange the contents of x and y. Therefore, no **return** statement is necessary.

The compiler performs type checking on the arguments to swap because the prototype declaration of swap includes argument types for each formal parameter. The identifiers within the parentheses of the declaration and definition can be the same or different. What is important is that the types of the actual arguments match those of the formal parameter lists in both the prototype declaration and the eventual definition.

7.4.2 Calls with a Variable Number of Arguments

To call a function with a variable number of arguments, simply specify any number of arguments in the function call. If there is a prototype declaration of the function, a variable number of arguments can be specified by placing a comma followed by three periods (**,...**) at the end of the formal parameter list or list of argument types (see Section 4.5, "Function Declarations"). The function call must include one argument for each type name declared in the formal parameter list or the list of argument type.

Similarly, the formal parameter list (or identifier list, in the obsolescent form) in the function definition can end with a comma followed by three periods (**,...**) to indicate a variable number of arguments. See Section 7.2, "Function Definitions," for more information about the form of the formal parameter list.

Note

To maintain compatibility with previous versions, a Microsoft extension to the ANSI C standard allows a comma without trailing periods (,) at the end of the list of formal parameters to indicate a variable number of arguments. See your User's Guide for information on enabling and disabling extensions.

All the arguments specified in the function call are placed on the stack. The number of formal parameters declared for the function determines how many of the arguments are taken from the stack and assigned to the formal parameters. You are responsible for retrieving any additional arguments from the stack and for determining how many arguments are present. See your User's Guide for information about macros that you can use to handle a variable number of arguments in a portable way.

7.4.3 Recursive Calls

Any function in a C program can be called recursively; that is, it can call itself. The C compiler allows any number of recursive calls to a function. Each time the function is called, new storage is allocated for the formal parameters and for the **auto** and **register** variables so that their values in previous, unfinished calls are not overwritten. Parameters are only directly accessible to the instance of the function in which they are created. Previous parameters are not directly accessible to ensuing instances of the function.

Note that variables declared with **static** storage do not require new storage with each recursive call. Their storage exists for the lifetime of the program. Each reference to such a variable accesses the same storage area.

Although the C compiler does not limit the number of times a function can be called recursively, the operating environment may impose a practical limit. Since each recursive call requires additional stack memory, too many recursive calls can cause a stack overflow.

Chapter 8
Preprocessor Directives and Pragmas

8.1 Introduction

A “preprocessor directive” is an instruction to the C preprocessor. The C preprocessor is a text processor that manipulates the text of a source file as the first phase of compilation. The compiler ordinarily invokes the preprocessor in its first pass, but the preprocessor can also be invoked separately to process text without compiling.

Preprocessor directives are typically used to make source programs easy to change and easy to compile in different execution environments. Directives in the source file tell the preprocessor to perform specific actions. For example, the preprocessor can replace tokens in the text, insert the contents of other files into the source file, or suppress compilation of part of the file by removing sections of text.

The C preprocessor recognizes the following directives:

#define	**#if**	**#line**
#elif	**#ifdef**	**#undef**
#else	**#ifndef**	
#endif	**#include**	

The number sign (#) must be the first non-white-space character on the line containing the directive; white-space characters can appear between the number sign and the first letter of the directive. Some directives include arguments or values. Any text that follows a directive (except an argument or value that is part of the directive) must be enclosed in comment delimiters (/* */).

Preprocessor directives can appear anywhere in a source file, but they apply only to the remainder of the source file in which they appear.

A “preprocessor operator” is an operator that is only recognized as an operator within the context of preprocessor directives. There are only three preprocessor-specific operators: the “stringizing” operator (#), the “token-pasting” (##) operator, and the **defined** operator. The first two are discussed in in the context of the **#define** directive in Sections 8.2.2.1 and 8.2.2.2. The **defined** operator is discussed in Section 8.4.1.

A “pragma” is a “pragmatic,” or practical, instruction to the C compiler. Pragmas in C source files are typically used to control the actions of the compiler in a particular portion of a program without affecting the program as a whole. Section 8.6 describes the syntax for pragmas. However, the compiler implementation defines the particular pragmas that are available and their meanings. See your User's Guide for information about the use and effects of pragmas.

8.2 Manifest Constants and Macros

The **#define** directive is typically used to associate meaningful identifiers with constants, keywords, and commonly used statements or expressions. Identifiers that represent constants are called "manifest constants." Identifiers that represent statements or expressions are called "macros."

Once you have defined an identifier, you cannot redefine it to a different value without first removing the original definition. However, you can redefine the identifier with exactly the same definition. Thus, the same definition can appear more than once in a program.

The **#undef** directive removes the definition of an identifier. Once you have removed the definition, you can redefine the identifier to a different value. Sections 8.2.2 and 8.2.3 discuss the **#define** and **#undef** directives, respectively.

In practical terms there are two types of macros. "Object-like" macros take no arguments, while "function-like" macros can be defined to accept arguments, so they look and act like function calls. Because macros do not generate actual function calls, you can make programs faster by replacing function calls with macros. However, macros can create problems if you do not define and use them with care. You may have to use parentheses in macro definitions with arguments to preserve the proper precedence in an expression. Also, macros may not handle expressions with side effects correctly. See the examples in Section 8.2.2 for more information.

8.2.1 Preprocessor Operators

There are three preprocessor-specific operators, one of which is represented by the number sign (#), one by a double number sign (##), and the third by the word **defined**. The # preceding an identifier in the body of a preprocessor macro allows strings to be formed when the macro is expanded. It is referred to as the "stringizing" operator. The ## operator, called the "token pasting" operator, allows tokens used as actual arguments to be concatenated to form other tokens. These two operators are used in the context of the **#define** directive and are described in Sections 2.2.2.1 and 2.2.2.2.

Finally, the **defined** operator simplifies the writing of compound expressions in certain macro directives. It is used in conditional compilation, and is therefore discussed in Section 8.4.1.

8.2.2 The #define Directive

Syntax

# define *identifier substitution-text*	Object-like macro
# define *identifier(parameter-list) substitution-text*	Function-like macro

The **#define** directive substitutes the *substitution-text* for all subsequent occurrences of the *identifier* in the source file. The *identifier* is replaced only when it forms a token. (Tokens are described in Chapter 2, "Elements of C" and Appendix B, "Syntax Summary.") For instance, the *identifier* is not replaced if it appears within a string or as part of a longer identifier.

If a *parameter-list* appears after the *identifier*, the **#define** directive replaces each occurrence of *identifier(parameter-list)* with a version of *substitution-text* that has actual arguments substituted for formal parameters.

The *substitution-text* consists of a series of tokens, such as keywords, constants, or complete statements. One or more white-space characters must separate the *substitution-text* from the *identifier* (or from the closing parenthesis of the *parameter-list*). This white space is not considered part of the substituted text, nor is any white space following the last token of the text. Text longer than one line can be continued onto the next line by placing a backslash (\) before the new-line character.

The *substitution-text* can also be empty. This option removes instances of the *identifier* from the source file. The *identifier* is still considered defined, however, and yields the value 1 when tested with the **#if** directive (discussed in Section 8.4.1).

The optional *parameter-list* consists of one or more formal parameter names separated by commas. Each name in the list must be unique, and the list must be enclosed in parentheses. No spaces can separate the *identifier* and the opening parenthesis. The scope of the formal parameter names extends to the new line that ends the *substitution-text.*

Formal parameter names appear in *substitution-text* to mark the places where actual values will be substituted. Each parameter name can appear more than once in the *substitution-text,* and the names can appear in any order.

The actual arguments following an instance of the *identifier* in the source file are matched to the formal parameters of the *parameter-list,* Each formal parameter in the *substitution-text* that is not preceded by a # or ## operator (or followed by a ## operator) is replaced by the corresponding actual argument. (These operators are described below in Sections 2.2.2.1 and 2.2.2.2.) The actual-argument list and the formal *parameter-list* must have the same number of arguments.

Any macros in the actual argument are expanded, and the expanded string is substituted for the formal parameter. However, if the name of the macro being defined occurs in the *substitution-text,* it is not expanded.

Arguments with side effects sometimes cause macros to produce unexpected results. A given formal parameter may appear more than once in a macro definition. If that formal parameter is replaced by an expression with side effects, the expression, with its side effects, is evaluated more than once (see example 4 below).

8.2.2.1 Stringizing Operator (#)

The # operator is used only with function-like macros. If the # precedes a formal parameter in the macro definition, the expanded actual argument passed by the macro invocation is treated as a string literal. The string literal then replaces each occurrence of the #-formal-parameter combination within the macro definition. Any white space between the tokens in the expanded actual argument is reduced to a single white space in the resulting string literal. Thus if a comment occurs between two tokens in the actual argument, it is reduced to a single white space. The parameter is automatically concatenated with any adjacent string literals from which it is sepatated only by white space. Furthermore, if a character passed as an argument to the macro would normally require an escape sequence when used in a string literal (for example the " character), the backslash is automatically inserted before the character. Example 6 illustrates some applications of the # operator.

Note

The Microsoft extension to the ANSI C standard that previously enabled expansion of macro formal arguments appearing in string literals and character constants is no longer supported. Macros that relied on this extension should be rewritten using the stringizing (#) operator.

8.2.2.2 Token-Pasting Operator (##)

The ## operator, referred to as the "token-pasting" or "concatenation" operator is used in both object-like and function-like macros. It permits joining together of separate tokens into a single token, and therefore cannot be the first or last token in the macro definition. Its use has the following form:

# **define** *identifier token# # token # # token ...*

The *identifier* represents the name by which the concatenated tokens will be known in the program before replacement. Each *token* represents a token defined elsewhere, either within the program or on the compiler command line. White space preceeding or following the operator is optional.

If a formal parameter in the macro definition is preceded or followed by the ## operator, the fomal parameter is immediately replaced by the unexpanded actual argument. The ## operator is then removed, and the tokens preceding and following it are concatenated. The resulting token must be a valid token, and it is then rescanned for possible replacement if it represents a macro name. Example 7 shows how tokens can be pasted together using the ## operator.

Examples

```
/******************** Example 1 ********************/

#define WIDTH       80
#define LENGTH      (WIDTH + 10)
```

The Example 1 defines the identifier WIDTH as the integer constant 80, and defines LENGTH in terms of WIDTH and the integer constant 10. Each occurrence of LENGTH is replaced by (WIDTH + 10). In turn, each occurrence of WIDTH + 10 is replaced by the expression (80 + 10). The parentheses around WIDTH + 10 are important because they control the interpretation in a statement such as the following:

```
var = LENGTH * 20;
```

After the preprocessing stage the statement becomes

```
var = (80 + 10) * 20;
```

which evaluates to 1800. Without parentheses, the result is

```
var = 80 + 10 * 20;
```

which evaluates to 280 because the multiplication operator (*) has higher precedence than the addition operator (+).

```
/******************** Example 2 ********************/

#define FILEMESSAGE "Attempt to create file \
failed because of insufficient space"
```

Example 2 defines the identifier FILEMESSAGE. The definition is extended to a second line by using the backslash escape character (\).

```
/******************** Example 3 ********************/

#define REG1        register
#define REG2        register
#define REG3
```

Example 3 defines three identifiers, REG1, REG2, and REG3. REG1 and REG2 are defined as the keyword **register**. The definition of REG3 is empty, so each occurrence of REG3 is removed from the source file. These directives can be used to ensure that the program's most important variables (declared with REG1 and REG2) are given **register** storage. (See the discussion of the **#if** directive in Section 8.4.1 for an expanded version of this example.)

```
/******************** Example 4 ********************/
```

```
#define MAX(x,y)      ((x) > (y)) ? (x) : (y)
```

Example 4 defines a macro named `MAX`. Each occurrence of the identifier `MAX` after the definition in the source file is replaced by the expression `((x) > (y)) ? (x) : (y)`, where actual values replace the parameters `x` and `y`. For example, the occurrence

```
MAX(1,2)
```

is replaced by

```
((1) > (2)) ? (1) : (2)
```

and the occurrence

```
MAX(i,s[i])
```

is replaced by

```
((i) > (s[i])) ? (i) : (s[i])
```

This macro is easier to read than the corresponding expression, which makes the source program easier to understand.

Note that arguments with side effects may cause this macro to produce unexpected results. For example, the occurrence `MAX(i, s[i++])` is replaced by `((i) > (s[i++])) ? (i) : (s[i++])`. The expression `(s[i++])` may be evaluated twice, so by the time the ternary expression has been fully evaluated, `i` has increased by 2. The result of the ternary expression is unpredictable, since its operands can be evaluated in any order, and the value of `i` varies depending on the evaluation order.

```
/******************** Example 5 ********************/

#define MULT(a,b)    ((a) * (b))
```

Example 5 defines the macro `MULT`. Once the macro is defined, an occurrence such as `MULT(3, 5)` is replaced by `(3) * (5)`. The parentheses around the parameters are important because they control the interpretation when complex expressions form the arguments to the macro. For instance, the occurrence `MULT(3 + 4, 5 + 6)` is replaced by `(3 + 4) * (5 + 6)`, which evaluates to 77. Without the parentheses, the result would be `3 + 4 * 5 + 6`, which evaluates to 29 because the multiplication operator (*) has higher precedence than the addition operator (+).

```
/******************** Example 6 ********************/

#define GREETING Hello, World!
#define show(x) printf(#x)

main()
{
        show(  x + z );
        printf("\n");
        show(n /* some comment */ + p);
        printf("\n");
        show(GREETING);
        printf("\n");
        show("This \" is a double quote mark");
        printf("\n");
        show('\x');
}
```

Example 6 defines two macros, one an object-like macro that expands to the string literal `Hello, world!`, and the other a function-like macro called `show`, that takes one argument. However, the definition of the second macro includes the stringizing operator (#) immediately preceding the formal parameter `x`. When an argument is passed to the `show` macro, the formal parameter is replaced by the expanded actual argument enclosed in double quotation marks, thus "stringizing" it.

As the preprocessor progresses through the source file, the references to `show` are expanded as follows:

`show(  x + z );` produces `printf("x + z");`

`show(n /* comment */ + p);` produces `printf("n + p");`

`show(GREETING);` produces `printf("Hello, world");`

`show("This \" is a double quote mark");`

produces

`printf("\"This \\\" is a double quote mark \"");`

and finally, `show('\x');` produces `printf("'\\x'");`

When the program is run, the screen output would be:

```
x + z
n + p
```

```
Hello, world
This " is a double quote mark
\x
```

```
/******************* Example 7 *********************/
#define father_ printf("functions look like this: foo()\n");
#define like_ printf("\n\nMicrosoft ");
#define son_ printf("macros can look like this: me_too()\n");
#define cat_tokens(x,y,z) x##y##z
#define father_like_son_ printf("C has them now!\n");
#define son_like_father_ printf("C has always had them\n");

main()
{
        like ;
        father ;
        like ;
        son ;
        cat_tokens(father_,like_,son_) ;
        cat_tokens(son_,like_,father_) ;
        like ;
        cat_tokens(son_,like_,father_) ;
        like ;
        cat_tokens(father_,like_,son_) ;
}
```

The tokens passed to the cat_tokens macro are pasted together to create other tokens defined elsewhere in the program. When this code is executed, the output will appear as follows:

```
Microsoft functions look like this: foo()
Microsoft macros can look like this: me_too()
C has them now!
C has always had them
Microsoft C has always had them
Microsoft C has them now!
```

8.2.3 The #undef Directive

Syntax

# **undef** *identifier*

The #**undef** directive removes the current definition of *identifier*. The preprocessor ignores subsequent occurrences of *identifier*. To remove a macro definition using #**undef**, give only the macro *identifier*; do not give a parameter list.

You can also apply the **#undef** directive to an identifier that has no previous definition. This ensures that the identifier is undefined.

The **#undef** directive is typically paired with a **#define** directive to create a region in a source program in which an identifier has a special meaning. For example, a specific function of the source program can use manifest constants to define environment-specific values that do not affect the rest of the program. The **#undef** directive also works with the **#if** directive (see Section 8.4.1) to control conditional compilation of the source program.

Example

```
#define WIDTH            80
#define ADD(X,Y)         (X) + (Y)
.
.
.
#undef WIDTH
#undef ADD
```

In this example, the **#undef** directive removes definitions of a manifest constant and a macro. Note that only the identifier of the macro is given.

8.3 Include Files

Syntax

# include "*pathname***"**
# include <*pathname***>**

The **#include** directive adds the contents of a given "include file" to another file. You can organize constant and macro definitions into include files and then use **#include** directives to add these definitions to any source file. Include files are also useful for incorporating declarations of external variables and complex data types. You only need to define and name the types once in an include file created for that purpose.

The **#include** directive tells the preprocessor to treat the contents of the named file as if they appeared in the source program at the point where the directive appears. The new text can also contain preprocessor directives. The preprocessor carries out directives in the new text, then continues processing the original text of the source file.

The *pathname* is a file name optionally preceded by a directory specification. It must name an existing file. The syntax of the file specification depends on the operating system on which the program is compiled.

The preprocessor uses the concept of a "standard" directory or directories to search for include files. The location of the standard directories for include files depends on the implementation and the operating system. See your User's Guide for a definition of the standard directories.

The preprocessor stops searching as soon as it finds a file with the given name. If you specify a complete, unambiguous path name for the include file, in double quotation marks (" "), the preprocessor searches only that path name and ignores the standard directories.

If you give an incomplete *pathname* enclosed in double quotation marks for the include file, the preprocessor first searches for the file in the same directory as the current source file (the "current working directory"); then in the directories specified on the compiler command line; and finally in the standard directories.

If the file specification is enclosed in angle brackets, the preprocessor does not search the current working directory. It begins by searching for the file in the directories specified on the compiler command line, then in the standard directories.

An **#include** directive can be nested; in other words, the directive can appear in a file named by another **#include** directive. When the preprocessor encounters the nested **#include** directive, it processes the named file and inserts it into the current file. The preprocessor uses the search procedures outlined above to search for nested include files.

The new file can also contain **#include** directives. Nesting can continue up to 10 levels. Once the nested **#include** is processed, the preprocessor continues to insert the enclosing include file into the original source file.

Examples

```
/******************** Example 1 ********************/
#include <stdio.h>       /* Example 1 */
```

Example 1 adds the contents of the file named `stdio.h` to the source program. The angle brackets cause the preprocessor to search the standard directories for `stdio.h`, after searching directories specified on the command line.

```
/******************** Example 2 ********************/
#include "defs.h"
```

Example 2 adds the contents of the file specified by `defs.h` to the source program. The double quotation marks mean that the preprocessor searches the directory containing the current source file first.

8.4 Conditional Compilation

This section describes the syntax and use of directives that control "conditional compilation." These directives allow you to suppress compilation of parts of a source file by testing a constant expression or identifier to determine which text blocks will be passed on to the compiler and which text blocks will be removed from the source file during preprocessing.

8.4.1 The #if, #elif, #else, and #endif Directives

Syntax

#if *restricted-constant-expression*
  ⟦ *substitution-text* ⟧
⟦ **#elif** *restricted-constant-expression*
  substitution-text ⟧
⟦ **#elif** *restricted-constant-expression*
  substitution-text ⟧
.
.
.

⟦ **#else**
  substitution-text ⟧
#endif

The **#if** directive, together with the **#elif**, **#else**, and **#endif** directives, controls compilation of portions of a source file. Each **#if** directive in a source file must be matched by a closing **#endif** directive. Any number of **#elif** directives can appear between the **#if** and **#endif** directives, but at most one **#else** directive is allowed. The **#else** directive, if present, must be the last directive before **#endif.**

The preprocessor selects one of the given blocks of *substitution-text* for further processing. A *substitution-text* block can be any sequence of text. It can occupy more than one line. Usually the *substitution-text* block is program text that has meaning to the compiler or the preprocessor. However, this is not a requirement; you can use the preprocessor to process any kind of text.

The preprocessor processes the selected *substitution-text* and passes it to the compiler. If the *substitution-text* contains preprocessor directives, the preprocessor carries out those directives.

Any substitution-text blocks not selected by the preprocessor are removed from the file during preprocessing. Thus, these text blocks are not compiled.

The preprocessor selects a single *substitution-text* block by evaluating the *restricted-constant-expressions* following each **#if** or **#elif** directive until it finds a true (nonzero) *restricted-constant-expression.* It selects all *substitution-text* between the first true *restricted-constant-expression* and the next number sign (#) which is not an **#elif** or **#else**.

If all occurrences of *restricted-constant-expression* are false, or if no **#elif** directives appear, the preprocessor selects the substitution-text after the **#else** clause. If the **#else** clause is omitted, and all *restricted-constant-expressions* in the **#if** block are false, no substitution text is selected.

Each *restricted-constant-expression* follows the rules for restricted constant expressions discussed in Section 5.2.10. Such expressions cannot contain **sizeof** expressions, type casts, or enumeration constants. However, they can contain the preprocessor operator **defined** in special constant expressions, as shown by the following syntax:

defined(*identifier***)**

This constant expression is considered true (nonzero) if the *identifier* is currently defined; otherwise, the condition is false (0). An identifier defined as empty text is considered defined.

The **#if**, **#elif**, **#else**, and **#endif** directives can nest in the text portions of other **#if** directives. Each nested **#else**, **#elif**, or **#endif** directive belongs to the closest preceding **#if** directive.

Examples

```
/******************** Example 1 ********************/

#if defined(CREDIT)
    credit();
#elif defined(DEBIT)
    debit();
#else
    printerror();
#endif
```

In example 1, the **#if** and **#endif** directives control compilation of one of three function calls. The function call to `credit` is compiled if the identifier CREDIT is defined. If the identifier DEBIT is defined, the function call to `debit` is compiled. If neither identifier is defined, the call to `printerror` is compiled. Note that CREDIT and `credit` are distinct identifiers in C because their cases are different.

```
/******************** Example 2 ********************/

#if DLEVEL > 5
    #define SIGNAL  1
    #if STACKUSE == 1
              #define STACK   200
    #else
              #define STACK   100
    #endif
#else
    #define SIGNAL  0
    #if STACKUSE == 1
              #define STACK   100
    #else
              #define STACK   50
    #endif
#endif
```

```
/******************** Example 3 ********************/

#if DLEVEL == 0
    #define STACK 0
#elif DLEVEL == 1
    #define STACK 100
#elif DLEVEL > 5
    display( debugptr );
#else
    #define STACK 200
#endif
```

Examples 2 and 3 assume a previously defined manifest constant named DLEVEL.

Example 2 shows two sets of nested **#if**, **#else**, and **#endif** directives. The first set of directives is processed only if DLEVEL > 5 is true. Otherwise, the second set is processed.

In Example 3, **#elif** and **#else** directives are used to make one of four choices, based on the value of DLEVEL. The manifest constant STACK is set to 0, 100, or 200, depending on the definition of DLEVEL. If DLEVEL is greater than 5, `display(debugptr);` is compiled and STACK is not defined.

```
/******************** Example 4 ********************/

#define REG1    register
#define REG2    register

#if defined(M_86)
    #define REG3
    #define REG4
    #define REG5
#else
    #define REG3    register
    #if defined(M_68000)
         #define REG4    register
         #define REG5    register
       #else
         #define REG4    register
         #define REG5
    #endif
#endif
```

Example 4 uses preprocessor directives to control the meaning of **register** declarations in a portable source file. The compiler assigns register storage to variables in the order in which the **register** declarations appear in the source file. If a program contains more **register** declarations than the machine allows, the compiler honors earlier declarations over later ones. The program may be less efficient if the variables declared later are more heavily used.

The definitions listed in example 4 can be used to give priority to the most important register declarations. REG1 and REG2 are defined as the **register** keyword to declare **register** storage for the two most important variables in the program. For example, in the following fragment, b and c have higher priority than a or d:

```
func(a)

REG3 int a;

{
        REG1 int b;
        REG2 int c;
        REG4 int d;
        .
        .
        .
}
```

When M_86 is defined, the preprocessor removes the REG3 identifier from the file by replacing it with empty text. This prevents a from receiving **register** storage at the expense of b and c. When M_68000 is defined, all four variables are declared to have **register** storage. When neither M_86 nor M_68000 is defined, a, b, and c are declared with **register** storage.

8.4.2 The #ifdef and #ifndef Directives

Syntax

#ifdef *identifier*
#ifndef *identifier*

The **#ifdef** and **#ifndef** directives perform the same task as the **#if** directive used with **defined**(*identifier*). You can use the **#ifdef** and **#ifndef** directives anywhere **#if** can be used. These directives are provided only for compatibility with previous versions of the language. The

defined(*identifier***)** constant expression used with the **#if** directive is preferred.

When the preprocessor encounters an **#ifdef** directive, it checks to see whether the *identifier* is currently defined. If so, the condition is true (nonzero); otherwise, the condition is false (0).

The **#ifndef** directive checks for the opposite of the condition checked by **#ifdef**. If the identifier has not been defined (or its definition has been removed with **#undef**), the condition is true (nonzero). Otherwise, the condition is false (0).

8.5 Line Control

Syntax

#line *constant* ⟦ **"***filename***"** ⟧

The **#line** directive tells the preprocessor to change the compiler's internally stored line number and file name to a given line number and file name. The compiler uses the line number and file name to refer to errors that it finds during compilation. The line number normally refers to the current input line, and the file name refers to the current input file. The line number is incremented after each line is processed.

If you change the line number and file name, the compiler ignores the previous values and to continues processing with the new values. The **#line** directive is typically used by program generators to cause error messages to rrefer to the original source file instead of the generated program.

The *constant* value in the **#line** directive can be any integer constant. The *filename* can be any combination of characters and must be enclosed in double quotation marks (" "). If *filename* is omitted, the previous file name remains unchanged.

The current line number and file name are always available through the predefined identifiers **__LINE__** and **__FILE__**. You can use the **__LINE__** and **__FILE__** identifiers to insert self-descriptive error messages into the program text.

The **__FILE__** identifier contains a string representing the file name, surrounded by double quotation marks (" "). Thus, you do not need to enclose the **__FILE__** identifier in quotation marks when you use it as a string.

Examples

```
/******************** Example 1 ********************/

#line 151 "copy.c"
```

In example 1, the internally stored line number is set to 151 and the file name is changed to `copy.c`.

```
/******************** Example 2 ********************/

#define ASSERT(cond)        if(!cond)\
{printf("assertion error line %d, file(%s)\n", \
__LINE__, __FILE__ );} else
```

In example 2, the macro ASSERT uses the predefined identifiers **__LINE__** and **__FILE__** to print an error message about the source file if a given "assertion" is not true. Note that no quotation marks are needed around **__FILE__**.

8.6 Pragmas

Syntax

# **pragma** *character-sequence*

A #**pragma** is an implementation-defined instruction to the compiler. The *character-sequence* is a series of characters that gives a specific compiler instruction and arguments, if any. The number sign (#) must be the first non-white-space character on the line containing the pragma; white-space characters can separate the number sign and the word **pragma**.

See your User's Guide for information about the pragmas available in your compiler implementation.

Appendix A Differences

This appendix summarizes differences between Microsoft C and the description of the C language found in Appendix A of *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, published in 1978 by Prentice-Hall, Inc. The following is a list of the differences, with cross-references to the corresponding section numbers in *The C Programming Language*:

Section Number in Kernighan and Ritchie	Microsoft C
2.2	Identifiers (including those used in preprocessor directives) are significant to 31 characters. External identifiers are also significant to 31 characters.
2.3	The identifiers **asm** and **entry** are no longer keywords. New keywords are **const**, **volatile**, **enum**, **signed**, and **void**. (The **volatile** keyword is implemented syntactically, but not semantically.) The identifiers **cdecl**, **far**, **fortran**, **huge**, **near**, and **pascal** may be keywords, depending on whether or not the corresponding options are enabled when a program is compiled (see your system documentation).
2.4.1	As a result of the method used to assign types to hexadecimal and octal constants, these constants always act like unsigned integers in type conversions.

2.4.3 Hexadecimal bit patterns consisting of a backslash (\), the letter **x**, and up to three hexadecimal digits are permitted as character constants (for example, \x012).

Microsoft C defines three additional escape sequences: **\v** represents a vertical tab (VT), \" represents the double-quote character, and **\a** represents the bell (also called alert).

Character constants always have type **int**, with the result that they are sign-extended in type conversions.

Adjacent quoted string literals are concatenated and treated as a single null-terminated string.

2.6 The **short** type is always 16 bits long, and the **long** type is 32 bits long. The size of an **int** is machine-dependent. On 8086/8088, 80186, and 80286 processors an **int** is 16 bits long, and on 80386 and 68000 processors it is 32 bits long.

4 The **char** type is signed by default, with the result that a **char** value is sign-extended in type conversions. (In some implementations, the default for the **char** type can be changed to unsigned at compile time.)

Two additional unsigned types are supported: **unsigned char** and **unsigned long**.

The keyword **unsigned** or **signed** can be applied as an adjective to an integer type. When **unsigned** appears alone, it means **unsigned int**. Similarly, when **signed** appears alone, it means **int**. The additional floating type **long double** is supported, but the **long float** type is no longer recognized. References to **long float** should be recoded to **double**. type

The type specifiers **const**and**volatile** can be used as modifiers for any fundamental, aggregate, or pointer type to indicate that the object or pointer value will not be modified. Both syntax and semantics of **const** are implemented, but only the syntax of volatile is implemented.

Microsoft C offers an additional fundamental type: the **enum** (enumeration) type. Variables of **enum** type are treated as integers in all cases. The keyword **void** has three different usages: as a function return-type specifier, it indicates that the function will not return a value. In an otherwise empty formal-parameter list, **void** means that no arguments will be passed. In the construction **void ***, it indicates a pointer to an object of unspecified type.

6.4 If the **near**, **far**, and **huge** keywords are enabled, pointers of different sizes may be used in a program. Operations with pointers of different sizes may cause conversion of pointers; the path of the conversion is implementation-defined.

6.6 The arithmetic conversions carried out by the Microsoft C Optimizing Compiler are outlined in Sections 5.3.1 and 5.7 of Chapter 5, "Expressions and Assignments." Although compatible with the Kernighan and Ritchie conversions, the Microsoft C conversions are described in greater detail, including the specific path for each type of conversion.

In addition to the usual arithmetic conversions, conversions between pointers of different sizes may be routinely carried out when the **near**, **far**, and **huge** keywords are enabled. The path of the pointer conversions is implementation-dependent.

7.2 In connection with the **sizeof** operator, a byte is defined as an 8-bit quantity.

7.14 A structure can be assigned to another structure of the same type.

8.2 The keywords **enum**, **const**, **volatile**, and **void** are additional type specifiers. The **volatile** keyword is implemented syntactically, but not semantically. The keyword **signed** or **unsigned** can serve either as a type specifier or as an adjective modifying an integer type.

Therefore, the following additional combinations are acceptable:

signed char
signed short
signed short int
signed long
signed long int
unsigned char
unsigned short
unsigned short int
unsigned long
unsigned long int

The **long float** type is not recognized. The **long double** type is recognized and treated in all instances the same as **double**.

8.4 The **const** and **volatile** keywords can be used to modify any fundamental, aggregate, or pointer object. The order of the type specifiers is not significant.

Optional formal-parameter lists or argument-type lists can be included in function declarations to notify the compiler of the number and types of arguments expected in a function call.

8.5 Bit fields can be declared to be any **signed** or **unsigned** integral type, except **enum**. However, in expressions bit fields are always treated as **unsigned**.

The names of structure and union members are not required to be distinct from structure and union tags or from the names of other variables.

No relationship exists between the members of two different structure types.

8.6 Unions can be initialized by giving a value for the first member of the union.

9.7 The *expression* of a **switch** can be any integral expression, but the value of the expression is always converted to an **int** type. An *expression* with **enum** type is permitted. Each of the **case** constant expressions is cast to the type of the *expression.*

10.1 New styles for function declarations and definition, as specified in the Draft Proposed American National Standard—Programming Language C , are completely supported. This includes the function prototype declaration, the prototype-style definition with formal parameters declared in the header, and the default creation of prototypes from the first reference to a function (if no explicit prototype is provided). The old function declaration and definition forms are also supported.

The formal parameter list in a function definition or declaration can end with a comma followed by three periods (,...) or just a comma (,) to indicate that the number of parameters is variable. The latter is supported only for compatibility with older versions of the compiler and should not be used in new code.

12 The number sign (#) introducing the preprocessor directive can be preceded by any combination of white-space characters. White space can also separate the number sign and the preprocessor keyword.

In addition to preprocessor directives, the source file can contain pragmas. Pragmas, like directives, are introduced by a number sign as the first non-white-space character in a line. The action defined by a particular pragma is implementation-dependent.

Three preprocessor-only operators are supported: the "stringizing" (#) operator, the concatenation or "token-pasting" (##) operator, and the **defined** operator.

12.3 The new combination **#if defined**(*identifier*) is intended to supplant the **#ifdef** and **#ifndef** directives. Use of the latter directives is discouraged.

	The new directive #**elif** (else-if) is designed for use in #**if** and #**if defined** blocks.
14.1	A structure or union can be assigned to another structure or union of the same type. Structures and unions can be passed by value to functions and returned by functions. In expressions involving −>, the expression preceding the arrow must have the same type (or must be cast to the same type) as the structure to which the member on the right-hand side of the arrow belongs.
17	The listed anachronisms are not recognized.

Appendix B
Syntax Summary

B.1 Tokens

keyword
identifier
constant
string
operator
separator

B.1.1 Keywords

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

* Semantics not yet implemented

The following identifiers may be keywords in some implementations. See your User's Guide for information.

cdecl
far
fortran
huge
near
pascal

B.1.2 Identifiers

identifier:
 letter
 underscore
 identifier letter
 identifier underscore
 identifier digit

letter–one of the following:
    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

underscore:
    _

digit–one of the following:
    0 1 2 3 4 5 6 7 8 9

B.1.3 Constants

constant:
    integer-constant
    long-constant
    floating-point-constant
    char-constant
    enum-constant

integer-constant:
    0
    decimal-constant
    octal-constant
    hexadecimal-constant

decimal-constant:
    nonzero-digit
    decimal-constant digit

nonzero-digit–one of the following:
    1 2 3 4 5 6 7 8 9

octal-constant:
    0*octal-digit*
    octal-constant octal-digit

octal-digit–one of the following:
    0 1 2 3 4 5 6 7

hexadecimal-constant:
    0x*hexadecimal-digit*
    0X*hexadecimal-digit*

hexadecimal-constant hexadecimal-digit

hexadecimal-digit–one of the following:
0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F

long-constant:
integer-constant l
integer-constant L

floating-point-constant:
fractional-constant exponent
fractional-constant
digit-seq exponent

fractional-constant:
digit-seq . *digit-seq*
. *digit-seq*
digit-seq .

digit-seq:
digit
digit-seq digit

exponent:
e *sign digit-seq*
E *sign digit-seq*
e *digit-seq*
E *digit-seq*

sign:
+
\–

char-constant:
'*char*'

char:
rep-char
escape-sequence

rep-char:
  Any single representable character except the single quote ('), backslash (\\), or new-line character

escape-sequence–one of the following:

```
\'     \"     \\       \d   \dd   \ddd
\xd    \xdd   \xddd    \a   \b    \f
\n     \r     \t       \v
```

enum-constant:
  identifier

B.1.4 Strings

string-literal:
  ""
  "*char-seq*"

char-seq:
  char
  char-seq char

B.1.5 Operators

operator–one of the following:

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

B.1.6 Separators

separator–one of the following:

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

B.2 Expressions

expression:
 identifier
 constant
 string
 expression(*expression-list*)
 expression(**void**)
 expression[*expression*]
 expression.*identifier*
 expression–>*identifier*
 unary-expression
 binary-expression
 ternary-expression
 assignment-expression
 (*expression*)
 (*type-name*)*expression*
 constant-expression

expression-list:
 expression
 expression-list , *expression*

unary-expression:
 unop expression
 sizeof(*expression*)

unop–one of the following:
 - ~ ! * &

lvalue:
 identifier
 expression[*expression*]
 expression.*expression*
 expression–>*expression*

> **expression*
> (*type-name*)*expression*
> (*lvalue*)

type-name:
> See Section B.3, "Declarations."

binary-expression:
> *expression binop expression*

binop–one of the following:

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

ternary-expression:
> *expression* **?** *expression* **:** *expression*

assignment-expression:
> *lvalue*++
> *lvalue*––
> ++*lvalue*
> ––*lvalue*
> *lvalue assignment-op expression*

assignment-op–one of the following:

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

constant-expression:
> *identifier*
> *constant*
> (*type-name*)*constant-expression*
> *unary-expression*
> *binary-expression*
> *ternary-expression*
> (*constant-expression*)

B.3 Declarations

declaration:
    sc-specifier type-specifier-list declarator-list;
    type-specifier-list declarator-list;
    sc-specifier declarator-list;
    typedef *type-specifier-list declarator-list*;

sc-specifier:
    auto
    extern
    register
    static

type-specifier:
    char
    double
    longdouble
    enum-specifier
    float
    int
    long
    short
    struct-specifier
    typedef-name
    union-specifier
    unsigned
    signed
    signed char
    const
    volatile

type-specifier-list:
    type-specifier
    type-specifier-listtype-specifier

    enum-specifier:
        enum *tag* { *enum-list*}
        enum { *enum-list*}
        enum *tag*

tag:
    identifier

enum-list:
 enumerator
 enum-list , *enumerator*

enumerator:
 identifier
 identifier = *constant-expression*

struct-specifier:
 struct *tag* { *member-declaration-list*}
 struct { *member-declaration-list*}
 struct *tag*

member-declaration-list:
 member-declaration
 member-declaration-list member-declaration

member-declaration:
 type-specifier declarator-list;
 type-specifier identifier : *constant-expression*;
 type-specifier : *constant-expression*;

declarator-list:
 declarator
 declarator = *initializer*
 declarator-list , *declarator*

declarator:
 identifier
 modifier-list identifier
 declarator[ ]
 declarator[*constant-expression*]
 * *declarator*
 declarator(**void**)
 declarator(*formal-parameter-list*)
 declarator(*arg-type-list*)
 (*declarator*)

modifier-list
 modifier
 modifier-list modifier

formal-parameter-list
 formal-parameter
 formal-parameter-list, *formal-parameter*

arg-type-list:
- *type-name*
- *arg-type-list*, *type-name*
- *arg-type-list*,...
- *arg-type-list*,
- **void**
- **void** *

type-name:
- *type-specifier*
- *type-specifier abstract-declarator*

abstract-declarator:
- *
- *modifier**
- [ ]
- (*arg-type-list*)
- **abstract-declarator*
- *abstract-declarator**
- *abstract-declarator*[ ]
- *abstract-declarator*[*constant-expression*]
- [ ]*abstract-declarator*
- [*constant-expression*]*abstract-declarator*
- *abstract-declarator*(**void**)
- *abstract-declarator*(*formal-parameter-list*)
- *abstract-declarator*(*arg-type-list*)
- (*abstract-declarator*)

initializer:
- *expression*
- {*initializer-list*}

initializer-list:
- *initializer*
- *initializer-list*, *initializer*

typedef-name:
- *identifier*

union-specifier:
- **union** *tag* {*member-declaration-list*}
- **union** {*member-declaration-list*}
- **union** *tag*

modifier:

cdecl
far
fortran
huge
near
pascal

modifier-list
modifier
modifier-list modifier

B.4 Statements

statement:
break;
case *constant-expression* **:** *statement*
compound-statement
continue;
default : *statement*
do *statement* **while(***expression***);**
*expression***;**
for (⟦*expression*⟧;⟦*expression*⟧;⟦*expression*⟧) *statement*;
goto *identifier*;
identifier **:** *statement*
if (*expression*) *statement* ⟦**else** *statement*⟧
;
return ⟦*expression*⟧**;**
switch (*expression*) *statement*
while (*expression*) *statement*

compound-statement:
{ ⟦*declaration-list*⟧⟦*statement-list*⟧ }

declaration-list:
declaration
declaration-list declaration

statement-list:
statement
statement-list statement

B.5 Definitions

definition:
function-definition
data-definition

function-definition:
⟦*sc-specifier*⟧ ⟦*type-specifier*⟧ *declarator* (⟦*formal-parameter-list*⟧) *compound-statement*
⟦*sc-specifier*⟧ ⟦*type-specifier*⟧ *declarator* (⟦*parameter-list*⟧) ⟦*parameter-decs*⟧ *compound-statement*

formal-parameter-list:
fixed-parameter-list
variable-parameter-list

parameter-list:
fixed-parameter-list
variable-parameter-list

fixed-parameter-list:
identifier
parameter-list , *identifier*

variable-parameter-list:
fixed-parameter-list,...
fixed-parameter-list,

parameter-decs:
declaration
declaration-list declaration

data-definition:
declaration

B.6 Preprocessor Directives

directive:
  #
  # **define** *identifier* ⟦(⟦*parameter-list*⟧)⟧ ⟦*token-seq*⟧
  # **elif** *restricted-constant-expression*
  # **else**
  # **endif**
  # **if** *restricted-constant-expression*
  # **ifdef** *identifier*
  # **ifndef** *identifier*
  # **include** "*string*"
  # **include** <*string*>
  # **line** *digit-seq*
  # **line** *digit-seq string*
  # **undef** *identifier*

token-seq:
  token
  token-seq token

restricted-constant-expression:
  defined (*identifier*)
  Any *constant-expression* except **sizeof** expressions, casts, and enumeration constants

B.7 Pragmas

pragma:
  # **pragma** *char-seq*

Language Reference Index